Subject: [PATCH v2 00/11] Request for Inclusion: kmem controller for memcg.
Posted by Glauber Costa on Thu, 09 Aug 2012 13:01:08 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.

The slab patches are also mature in my self evaluation and could be merged not too long after this. For the reference, the last discussion about them happened at http://lwn.net/Articles/508087/

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

git://github.com/glommer/linux.git kmemcg-slab

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 (9):
  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
  memcg: disable kmem code when not in use.
  memcg: propagate kmem limiting information to children
  memcg: allow a memcg with kmem charges to be destructed.
  protect architectures where THREAD_SIZE >= PAGE_SIZE against fork
    bombs

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

 include/linux/gfp.h | 10 +-
 include/linux/memcontrol.h | 82 ++++++++
 include/linux/thread_info.h | 2 +
 kernel/fork.c | 4 +-
 mm/memcontrol.c | 443 ++++++++++++++++++++++++++++++++++++++---
 mm/page_alloc.c | 38 ++++
 6 files changed, 546 insertions(+), 33 deletions(-)

--
1.7.11.2

Subject: [PATCH v2 01/11] memcg: Make it possible to use the stock for more than one page.
Posted by Glauber Costa on Thu, 09 Aug 2012 13:01:09 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>
---
 mm/memcontrol.c | 28 ++++++++++++++++++++----------
 1 file changed, 18 insertions(+), 10 deletions(-)

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 95162c9..bc7bfa7 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -2096,20 +2096,28 @@ 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);
@@ -2408,7 +2416,7 @@ 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 {
@@ -2433,7 +2441,7 @@ 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
--
1.7.11.2
```

Subject: [PATCH v2 02/11] memcg: Reclaim when more than one page needed.
Posted by Glauber Costa on Thu, 09 Aug 2012 13:01:10 GMT

From: Suleiman Souhlal <ssouhlal@FreeBSD.org>

mem_cgroup_do_charge() was written before kmem accounting, and expects
three cases: being called for 1 page, being called for a stock of 32
pages, or being called for a hugepage. If we call for 2 or 3 pages (and
both the stack and several slabs used in process creation are such, at
least with the debug options I had), it assumed it's being called for
stock and just retried without reclaiming.

Fix that by passing down a minsize argument in addition to the csize.

And what to do about that (csize == PAGE_SIZE && ret) retry? If it's
needed at all (and presumably is since it's there, perhaps to handle
races), then it should be extended to more than PAGE_SIZE, yet how far?
And should there be a retry count limit, of what? For now retry up to
COSTLY_ORDER (as page_alloc.c does) and make sure not to do it if
__GFP_NORETRY.

[v4: fixed nr pages calculation pointed out by Christoph Lameter]

Signed-off-by: Suleiman Souhlal <suleiman@google.com>
Signed-off-by: Glauber Costa <glommer@parallels.com>
Reviewed-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
---
 mm/memcontrol.c | 16 +++++++++-------
 1 file changed, 9 insertions(+), 7 deletions(-)

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index bc7bfa7..2cef99a 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -2294,7 +2294,8 @@ enum {
 };

 static int mem_cgroup_do_charge(struct mem_cgroup *memcg, gfp_t gfp_mask,
-    unsigned int nr_pages, bool oom_check)
+   unsigned int nr_pages, unsigned int min_pages,
+   bool oom_check)
 {
  unsigned long csize = nr_pages * PAGE_SIZE;
  struct mem_cgroup *mem_over_limit;
@@ -2317,18 +2318,18 @@ static int mem_cgroup_do_charge(struct mem_cgroup *memcg,
gfp_t gfp_mask,
  } else
   mem_over_limit = mem_cgroup_from_res_counter(fail_res, res);
  /*
- * nr_pages can be either a huge page (HPAGE_PMD_NR), a batch

```
-  * of regular pages (CHARGE_BATCH), or a single regular page (1).
-  *
   * Never reclaim on behalf of optional batching, retry with a
   * single page instead.
   */
- if (nr_pages == CHARGE_BATCH)
+ if (nr_pages > min_pages)
   return CHARGE_RETRY;

  if (!(gfp_mask & __GFP_WAIT))
   return CHARGE_WOULDBLOCK;

+ if (gfp_mask & __GFP_NORETRY)
+  return CHARGE_NOMEM;
+
  ret = mem_cgroup_reclaim(mem_over_limit, gfp_mask, flags);
  if (mem_cgroup_margin(mem_over_limit) >= nr_pages)
   return CHARGE_RETRY;
@@ -2341,7 +2342,7 @@ static int mem_cgroup_do_charge(struct mem_cgroup *memcg, gfp_t
gfp_mask,
   * unlikely to succeed so close to the limit, and we fall back
   * to regular pages anyway in case of failure.
   */
- if (nr_pages == 1 && ret)
+ if (nr_pages <= (1 << PAGE_ALLOC_COSTLY_ORDER) && ret)
   return CHARGE_RETRY;

  /*
@@ -2476,7 +2477,8 @@ again:
    nr_oom_retries = MEM_CGROUP_RECLAIM_RETRIES;
   }

-  ret = mem_cgroup_do_charge(memcg, gfp_mask, batch, oom_check);
+  ret = mem_cgroup_do_charge(memcg, gfp_mask, batch, nr_pages,
+      oom_check);
   switch (ret) {
   case CHARGE_OK:
    break;
--
1.7.11.2
```

Subject: [PATCH v2 03/11] memcg: change defines to an enum
Posted by Glauber Costa on Thu, 09 Aug 2012 13:01:11 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>
CC: Michal Hocko <mhocko@suse.cz>
CC: Johannes Weiner <hannes@cmpxchg.org>
Acked-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
---
 mm/memcontrol.c | 26 ++++++++++++++++++----------
 1 file changed, 16 insertions(+), 10 deletions(-)

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 2cef99a..b0e29f4 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -393,9 +393,12 @@ 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)
@@ -3983,7 +3986,8 @@ 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);
@@ -4019,7 +4023,8 @@ 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;

@@ -4095,7 +4100,8 @@ 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);
@@ -4423,7 +4429,7 @@ 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;

@@ -4506,7 +4512,7 @@ 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;

@@ -4584,7 +4590,7 @@ 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);
@@ -4609,7 +4615,7 @@ 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);
```

--
1.7.11.2

Subject: [PATCH v2 04/11] kmem accounting basic infrastructure
Posted by Glauber Costa on Thu, 09 Aug 2012 13:01:12 GMT
View Forum Message <> Reply to Message

This patch adds the basic infrastructure for the accounting of the slab caches. 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.

The code is not enabled until a limit is set. This can be tested by the flag "kmem_accounted". This means that after the patch is applied, no behavioral changes exists for whoever is still using memcg to control their memory usage.

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)

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Michal Hocko <mhocko@suse.cz>
CC: Johannes Weiner <hannes@cmpxchg.org>
Reviewed-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
---
 mm/memcontrol.c | 69
++-
 1 file changed, 68 insertions(+), 1 deletion(-)

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index b0e29f4..54e93de 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c

```
@@ -273,6 +273,10 @@ 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.
  */
@@ -287,6 +291,7 @@ struct mem_cgroup {
  * Should the accounting and control be hierarchical, per subtree?
  */
 bool use_hierarchy;
+bool kmem_accounted;

 bool  oom_lock;
 atomic_t under_oom;
@@ -397,6 +402,7 @@ enum res_type {
 _MEM,
 _MEMSWAP,
 _OOM_TYPE,
+_KMEM,
};

#define MEMFILE_PRIVATE(x, val) ((x) << 16 | (val))
@@ -1499,6 +1505,10 @@ 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));

 mem_cgroup_print_oom_stat(memcg);
}
@@ -4008,6 +4018,9 @@ 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();
 }
```

```
@@ -4046,8 +4059,23 @@ 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 = res_counter_set_limit(&memcg->kmem, val);
+   if (ret)
+    break;
+   /*
+    * Once enabled, can't be disabled. We could in theory
+    * disable it if we haven't yet created any caches, or
+    * if we can shrink them all to death.
+    *
+    * But it is not worth the trouble
+    */
+   if (!memcg->kmem_accounted && val != RESOURCE_MAX)
+    memcg->kmem_accounted = true;
+  } else
+   return -EINVAL;
   break;
  case RES_SOFT_LIMIT:
   ret = res_counter_memparse_write_strategy(buffer, &val);
@@ -4113,12 +4141,16 @@ 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;
@@ -4672,6 +4704,33 @@ static int mem_cgroup_oom_control_write(struct cgroup *cgrp,
 }

 #ifdef CONFIG_MEMCG_KMEM
+static struct cftype kmem_cgroup_files[] = {
```

```
+ {
+  .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,
+ },
+ {},
+};
+
 static int memcg_init_kmem(struct mem_cgroup *memcg, struct cgroup_subsys *ss)
 {
  return mem_cgroup_sockets_init(memcg, ss);
@@ -5015,6 +5074,12 @@ mem_cgroup_create(struct cgroup *cont)
   int cpu;
   enable_swap_cgroup();
   parent = NULL;
+
+#ifdef CONFIG_MEMCG_KMEM
+  WARN_ON(cgroup_add_cftypes(&mem_cgroup_subsys,
+        kmem_cgroup_files));
+#endif
+
   if (mem_cgroup_soft_limit_tree_init())
    goto free_out;
   root_mem_cgroup = memcg;
@@ -5033,6 +5098,7 @@ 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.
@@ -5043,6 +5109,7 @@ mem_cgroup_create(struct cgroup *cont)
  } else {
   res_counter_init(&memcg->res, NULL);
   res_counter_init(&memcg->memsw, NULL);
+  res_counter_init(&memcg->kmem, NULL);
  }
  memcg->last_scanned_node = MAX_NUMNODES;
  INIT_LIST_HEAD(&memcg->oom_notify);
--
1.7.11.2
```

Subject: [PATCH v2 05/11] Add a __GFP_KMEMCG flag
Posted by Glauber Costa on Thu, 09 Aug 2012 13:01:13 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.

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
CC: Michal Hocko <mhocko@suse.cz>
CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Johannes Weiner <hannes@cmpxchg.org>
CC: Suleiman Souhlal <suleiman@google.com>
CC: Rik van Riel <riel@redhat.com>
CC: Mel Gorman <mel@csn.ul.ie>
---
 include/linux/gfp.h | 7 ++++++-
 1 file changed, 6 insertions(+), 1 deletion(-)

```
diff --git a/include/linux/gfp.h b/include/linux/gfp.h
index f9bc873..d8eae4d 100644
--- a/include/linux/gfp.h
+++ b/include/linux/gfp.h
@@ -35,6 +35,11 @@ struct vm_area_struct;
 #else
 #define ___GFP_NOTRACK  0
 #endif
+#ifdef CONFIG_MEMCG_KMEM
+#define ___GFP_KMEMCG  0x400000u
+#else
+#define ___GFP_KMEMCG  0
```

```
+#endif
 #define ___GFP_OTHER_NODE 0x800000u
 #define ___GFP_WRITE  0x1000000u

@@ -91,7 +96,7 @@ 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).
--
1.7.11.2
```

Subject: [PATCH v2 06/11] memcg: kmem controller infrastructure
Posted by Glauber Costa on Thu, 09 Aug 2012 13:01:14 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 accessors. 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.

[v2: improved comments and standardized function names]

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
CC: Michal Hocko <mhocko@suse.cz>
CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Johannes Weiner <hannes@cmpxchg.org>
---
 include/linux/memcontrol.h | 79 ++++++++++++++++++++++
 mm/memcontrol.c | 185 +++
 2 files changed, 264 insertions(+)

diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
index 8d9489f..75b247e 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 +400,11 @@ struct sock;
 #ifdef CONFIG_MEMCG_KMEM
 void sock_update_memcg(struct sock *sk);
 void sock_release_memcg(struct sock *sk);
+
+#define memcg_kmem_on 1
+bool __memcg_kmem_new_page(gfp_t gfp, void *handle, int order);
+void __memcg_kmem_commit_page(struct page *page, void *handle, int order);
+void __memcg_kmem_free_page(struct page *page, int order);
 #else
 static inline void sock_update_memcg(struct sock *sk)
 {
@@ -406,6 +412,79 @@ static inline void sock_update_memcg(struct sock *sk)
 static inline void sock_release_memcg(struct sock *sk)
 {
 }
+
+#define memcg_kmem_on 0
+static inline bool
+__memcg_kmem_new_page(gfp_t gfp, void *handle, int order)
+{
+ return false;
+}
+
+static inline void  __memcg_kmem_free_page(struct page *page, int order)
+{
+}
+
+static inline void
+__memcg_kmem_commit_page(struct page *page, struct mem_cgroup *handle, int order)
+{
+}
 #endif /* CONFIG_MEMCG_KMEM */
+
+/**
+ * memcg_kmem_new_page: verify if a new kmem allocation is allowed.
+ * @gfp: the gfp allocation flags.
+ * @handle: 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_new_page(gfp_t gfp, void *handle, int order)
+{
+ if (!memcg_kmem_on)
+  return true;
+ if (!(gfp & __GFP_KMEMCG) || (gfp & __GFP_NOFAIL))
+  return true;
+ if (in_interrupt() || (!current->mm) || (current->flags & PF_KTHREAD))
+  return true;
+ return __memcg_kmem_new_page(gfp, handle, order);
+}
+
+/**
+ * memcg_kmem_free_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_free_page(struct page *page, int order)
+{
+ if (memcg_kmem_on)
+  __memcg_kmem_free_page(page, order);
+}
+
+/**
+ * memcg_kmem_commit_page: embeds correct memcg in a page
+ * @handle: a pointer to the memcg this was charged against.
+ * @page: pointer to struct page recently allocated
+ * @handle: the memcg structure we charged against
+ * @order: allocation order.
+ *
+ * Needs to be called after memcg_kmem_new_page, 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 @handle to the
+ * corresponding page_cgroup.
+ */
+static __always_inline void
+memcg_kmem_commit_page(struct page *page, struct mem_cgroup *handle, int order)
+{
+ if (memcg_kmem_on)
```

```
+  __memcg_kmem_commit_page(page, handle, order);
+}
 #endif /* _LINUX_MEMCONTROL_H */

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 54e93de..e9824c1 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
@@ -434,6 +438,9 @@ struct mem_cgroup *mem_cgroup_from_css(struct cgroup_subsys_state
*s)
 #include <net/ip.h>

 static bool mem_cgroup_is_root(struct mem_cgroup *memcg);
+static int memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t gfp, s64 delta);
+static void memcg_uncharge_kmem(struct mem_cgroup *memcg, s64 delta);
+
 void sock_update_memcg(struct sock *sk)
 {
  if (mem_cgroup_sockets_enabled) {
@@ -488,6 +495,118 @@ struct cg_proto *tcp_proto_cgroup(struct mem_cgroup *memcg)
 }
 EXPORT_SYMBOL(tcp_proto_cgroup);
 #endif /* CONFIG_INET */
+
+static inline bool memcg_kmem_enabled(struct mem_cgroup *memcg)
+{
+ return !mem_cgroup_disabled() && !mem_cgroup_is_root(memcg) &&
+  memcg->kmem_accounted;
+}
+
+/*
+ * 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
+ *
+ * Returning true means the allocation is possible.
+ */
+bool __memcg_kmem_new_page(gfp_t gfp, void *_handle, int order)
+{
+ struct mem_cgroup *memcg;
+ struct mem_cgroup **handle = (struct mem_cgroup **)_handle;
+ bool ret = true;
+ size_t size;
+ struct task_struct *p;
+
+ *handle = NULL;
+ rcu_read_lock();
+ p = rcu_dereference(current->mm->owner);
+ memcg = mem_cgroup_from_task(p);
+ if (!memcg_kmem_enabled(memcg))
+  goto out;
+
+ mem_cgroup_get(memcg);
+
+ size = PAGE_SIZE << order;
+ ret = memcg_charge_kmem(memcg, gfp, size) == 0;
+ if (!ret) {
+  mem_cgroup_put(memcg);
+  goto out;
+ }
+
+ *handle = memcg;
+out:
+ rcu_read_unlock();
+ return ret;
+}
+EXPORT_SYMBOL(__memcg_kmem_new_page);
+
+void __memcg_kmem_commit_page(struct page *page, void *handle, int order)
+{
+ struct page_cgroup *pc;
+ struct mem_cgroup *memcg = handle;
+
+ if (!memcg)
+  return;
+
+ WARN_ON(mem_cgroup_is_root(memcg));
+ /* The page allocation must have failed. Revert */
+ if (!page) {
+  size_t size = PAGE_SIZE << order;
```

```
+
+  memcg_uncharge_kmem(memcg, size);
+  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_free_page(struct page *page, int order)
+{
+ struct mem_cgroup *memcg;
+ size_t size;
+ struct page_cgroup *pc;
+
+ if (mem_cgroup_disabled())
+  return;
+
+ pc = lookup_page_cgroup(page);
+ lock_page_cgroup(pc);
+ memcg = pc->mem_cgroup;
+ pc->mem_cgroup = NULL;
+ if (!PageCgroupUsed(pc)) {
+  unlock_page_cgroup(pc);
+  return;
+ }
+ ClearPageCgroupUsed(pc);
+ unlock_page_cgroup(pc);
+
+ /*
+  * Checking if kmem accounted is enabled won't work for uncharge, since
+  * it is possible that the user enabled kmem tracking, allocated, and
+  * then disabled it again.
+  *
+  * We trust if there is a memcg associated with the page, it is a valid
+  * allocation
+  */
+ if (!memcg)
+  return;
+
+ WARN_ON(mem_cgroup_is_root(memcg));
+ size = (1 << order) << PAGE_SHIFT;
+ memcg_uncharge_kmem(memcg, size);
+ mem_cgroup_put(memcg);
```

```
+}
+EXPORT_SYMBOL(__memcg_kmem_free_page);
 #endif /* CONFIG_MEMCG_KMEM */

 #if defined(CONFIG_INET) && defined(CONFIG_MEMCG_KMEM)
@@ -5759,3 +5878,69 @@ static int __init enable_swap_account(char *s)
 __setup("swapaccount=", enable_swap_account);

 #endif
+
+#ifdef CONFIG_MEMCG_KMEM
+int memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t gfp, s64 delta)
+{
+ struct res_counter *fail_res;
+ struct mem_cgroup *_memcg;
+ int ret;
+ bool may_oom;
+ bool nofail = false;
+
+ may_oom = (gfp & __GFP_WAIT) && (gfp & __GFP_FS) &&
+     !(gfp & __GFP_NORETRY);
+
+ ret = 0;
+
+ if (!memcg)
+  return ret;
+
+ _memcg = memcg;
+ ret = __mem_cgroup_try_charge(NULL, gfp, delta / PAGE_SIZE,
+     &_memcg, may_oom);
+
+ if (ret == -EINTR)  {
+  nofail = true;
+  /*
+   * __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
+   */
+  res_counter_charge_nofail(&memcg->res, delta, &fail_res);
+  if (do_swap_account)
+   res_counter_charge_nofail(&memcg->memsw, delta,
+       &fail_res);
+  ret = 0;
+ } else if (ret == -ENOMEM)
```

```
+  return ret;
+
+ if (nofail)
+  res_counter_charge_nofail(&memcg->kmem, delta, &fail_res);
+ else
+  ret = res_counter_charge(&memcg->kmem, delta, &fail_res);
+
+ if (ret) {
+  res_counter_uncharge(&memcg->res, delta);
+  if (do_swap_account)
+   res_counter_uncharge(&memcg->memsw, delta);
+ }
+
+ return ret;
+}
+
+void memcg_uncharge_kmem(struct mem_cgroup *memcg, s64 delta)
+{
+ if (!memcg)
+  return;
+
+ res_counter_uncharge(&memcg->kmem, delta);
+ res_counter_uncharge(&memcg->res, delta);
+ if (do_swap_account)
+  res_counter_uncharge(&memcg->memsw, delta);
+}
+#endif /* CONFIG_MEMCG_KMEM */
--
1.7.11.2
```

Subject: [PATCH v2 07/11] mm: Allocate kernel pages to the right memcg
Posted by Glauber Costa on Thu, 09 Aug 2012 13:01:15 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().

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Christoph Lameter <cl@linux.com>

```
CC: Pekka Enberg <penberg@cs.helsinki.fi>
CC: Michal Hocko <mhocko@suse.cz>
CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Johannes Weiner <hannes@cmpxchg.org>
CC: Suleiman Souhlal <suleiman@google.com>
---
 include/linux/gfp.h |  3 +++
 mm/page_alloc.c     | 38 ++++++++++++++++++++++++++++++++++++++
 2 files changed, 41 insertions(+)

diff --git a/include/linux/gfp.h b/include/linux/gfp.h
index d8eae4d..029570f 100644
--- a/include/linux/gfp.h
+++ b/include/linux/gfp.h
@@ -370,6 +370,9 @@ 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 b956cec..da341dc 100644
--- a/mm/page_alloc.c
+++ b/mm/page_alloc.c
@@ -2532,6 +2532,7 @@ __alloc_pages_nodemask(gfp_t gfp_mask, unsigned int order,
  struct page *page = NULL;
  int migratetype = allocflags_to_migratetype(gfp_mask);
  unsigned int cpuset_mems_cookie;
+ void *handle = NULL;

  gfp_mask &= gfp_allowed_mask;

@@ -2543,6 +2544,13 @@ __alloc_pages_nodemask(gfp_t gfp_mask, unsigned int order,
   return NULL;

  /*
+  * Will only have any effect when __GFP_KMEMCG is set.
+  * This is verified in the (always inline) callee
+  */
+ if (!memcg_kmem_new_page(gfp_mask, &handle, order))
+  return NULL;
+
+ /*
   * Check the zones suitable for the gfp_mask contain at least one
```

```
   * valid zone. It's possible to have an empty zonelist as a result
   * of GFP_THISNODE and a memoryless node
@@ -2583,6 +2591,8 @@ out:
  if (unlikely(!put_mems_allowed(cpuset_mems_cookie) && !page))
   goto retry_cpuset;

+ memcg_kmem_commit_page(page, handle, order);
+
  return page;
 }
 EXPORT_SYMBOL(__alloc_pages_nodemask);
@@ -2635,6 +2645,34 @@ 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_free_page(page, order);
+ __free_pages(page, order);
+}
+EXPORT_SYMBOL(__free_accounted_pages);
+
+void free_accounted_pages(unsigned long addr, unsigned int order)
+{
+ if (addr != 0) {
+  VM_BUG_ON(!virt_addr_valid((void *)addr));
+  memcg_kmem_free_page(virt_to_page((void *)addr), order);
+  __free_pages(virt_to_page((void *)addr), order);
+ }
+}
+EXPORT_SYMBOL(free_accounted_pages);
+
 static void *make_alloc_exact(unsigned long addr, unsigned order, size_t size)
 {
  if (addr) {
--
1.7.11.2
```

Subject: [PATCH v2 08/11] memcg: disable kmem code when not in use.
Posted by Glauber Costa on Thu, 09 Aug 2012 13:01:16 GMT
View Forum Message <> Reply to Message

We can use jump labels to patch the code in or out when not used.

Because the assignment: memcg->kmem_accounted = true is done after the
jump labels increment, we guarantee that the root memcg will always be
selected until all call sites are patched (see memcg_kmem_enabled).
This guarantees that no mischarges are applied.

Jump label decrement happens when the last reference count from the
memcg dies. This will only happen when the caches are all dead.

Signed-off-by: Glauber Costa <glommer@parallels.com>
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: Johannes Weiner <hannes@cmpxchg.org>
CC: Suleiman Souhlal <suleiman@google.com>
---
 include/linux/memcontrol.h | 5 ++++-
 mm/memcontrol.c | 50 ++----------
 2 files changed, 44 insertions(+), 11 deletions(-)

diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
index 75b247e..f39d933 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;
@@ -401,7 +402,9 @@ struct sock;
 void sock_update_memcg(struct sock *sk);
 void sock_release_memcg(struct sock *sk);

-#define memcg_kmem_on 1
+extern struct static_key memcg_kmem_enabled_key;
+#define memcg_kmem_on static_key_false(&memcg_kmem_enabled_key)
+
 bool __memcg_kmem_new_page(gfp_t gfp, void *handle, int order);
 void __memcg_kmem_commit_page(struct page *page, void *handle, int order);
 void __memcg_kmem_free_page(struct page *page, int order);

```
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index e9824c1..3216292 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -437,6 +437,10 @@ struct mem_cgroup *mem_cgroup_from_css(struct
cgroup_subsys_state *s)
 #include <net/sock.h>
 #include <net/ip.h>

+struct static_key memcg_kmem_enabled_key;
+/* so modules can inline the checks */
+EXPORT_SYMBOL(memcg_kmem_enabled_key);
+
 static bool mem_cgroup_is_root(struct mem_cgroup *memcg);
 static int memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t gfp, s64 delta);
 static void memcg_uncharge_kmem(struct mem_cgroup *memcg, s64 delta);
@@ -607,6 +611,16 @@ void __memcg_kmem_free_page(struct page *page, int order)
  mem_cgroup_put(memcg);
 }
 EXPORT_SYMBOL(__memcg_kmem_free_page);
+
+static void disarm_kmem_keys(struct mem_cgroup *memcg)
+{
+ if (memcg->kmem_accounted)
+  static_key_slow_dec(&memcg_kmem_enabled_key);
+}
+#else
+static void disarm_kmem_keys(struct mem_cgroup *memcg)
+{
+}
 #endif /* CONFIG_MEMCG_KMEM */

 #if defined(CONFIG_INET) && defined(CONFIG_MEMCG_KMEM)
@@ -622,6 +636,12 @@ static void disarm_sock_keys(struct mem_cgroup *memcg)
 }
 #endif

+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 *
@@ -4147,6 +4167,24 @@ 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 void memcg_update_kmem_limit(struct mem_cgroup *memcg, u64 val)
+{
+#ifdef CONFIG_MEMCG_KMEM
+ /*
+  * Once enabled, can't be disabled. We could in theory disable it if we
+  * haven't yet created any caches, or if we can shrink them all to
+  * death. But it is not worth the trouble.
+  */
+ mutex_lock(&set_limit_mutex);
+ if (!memcg->kmem_accounted && val != RESOURCE_MAX) {
+  static_key_slow_inc(&memcg_kmem_enabled_key);
+  memcg->kmem_accounted = true;
+ }
+ mutex_unlock(&set_limit_mutex);
+#endif
+}
+
 /*
  * The user of this function is...
  * RES_LIMIT.
@@ -4184,15 +4222,7 @@ static int mem_cgroup_write(struct cgroup *cont, struct cftype *cft,
    ret = res_counter_set_limit(&memcg->kmem, val);
    if (ret)
     break;
-   /*
-    * Once enabled, can't be disabled. We could in theory
-    * disable it if we haven't yet created any caches, or
-    * if we can shrink them all to death.
-    *
-    * But it is not worth the trouble
-    */
-   if (!memcg->kmem_accounted && val != RESOURCE_MAX)
-    memcg->kmem_accounted = true;
+   memcg_update_kmem_limit(memcg, val);
   } else
    return -EINVAL;
   break;
@@ -5054,7 +5084,7 @@ 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
--
1.7.11.2
```

Subject: [PATCH v2 09/11] memcg: propagate kmem limiting information to children
Posted by Glauber Costa on Thu, 09 Aug 2012 13:01:17 GMT
View Forum Message <> Reply to Message

The current memcg slab cache management fails to present satisfatory
hierarchical behavior in the following scenario:

-> /cgroups/memory/A/B/C

* kmem limit set at A,
* A and B have no tasks,
* span a new task in in C.

Because kmem_accounted is a boolean that was not set for C, no
accounting would be done. This is, however, not what we expect.

The basic idea, is that when a cgroup is limited, we walk the tree
upwards (something Kame and I already thought about doing for other
purposes), and make sure that we store the information about the parent
being limited in kmem_accounted (that is turned into a bitmap: two
booleans would not be space efficient). The code for that is taken from
sched/core.c. My reasons for not putting it into a common place is to
dodge the type issues that would arise from a common implementation
between memcg and the scheduler - but I think that it should ultimately
happen, so if you want me to do it now, let me know.

We do the reverse operation when a formerly limited cgroup becomes
unlimited.

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
CC: Michal Hocko <mhocko@suse.cz>
CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Johannes Weiner <hannes@cmpxchg.org>
CC: Suleiman Souhlal <suleiman@google.com>
---
 mm/memcontrol.c | 88 ++------
 1 file changed, 79 insertions(+), 9 deletions(-)

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 3216292..3d30b79 100644

```
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -295,7 +295,8 @@ struct mem_cgroup {
   * Should the accounting and control be hierarchical, per subtree?
   */
  bool use_hierarchy;
- bool kmem_accounted;
+
+ unsigned long kmem_accounted; /* See KMEM_ACCOUNTED_*, below */

  bool  oom_lock;
  atomic_t under_oom;
@@ -348,6 +349,38 @@ struct mem_cgroup {
 #endif
 };

+enum {
+ KMEM_ACCOUNTED_THIS, /* accounted by this cgroup itself */
+ KMEM_ACCOUNTED_PARENT, /* accounted by any of its parents. */
+};
+
+#ifdef CONFIG_MEMCG_KMEM
+static bool memcg_kmem_account(struct mem_cgroup *memcg)
+{
+ return !test_and_set_bit(KMEM_ACCOUNTED_THIS, &memcg->kmem_accounted);
+}
+
+static bool memcg_kmem_clear_account(struct mem_cgroup *memcg)
+{
+ return test_and_clear_bit(KMEM_ACCOUNTED_THIS, &memcg->kmem_accounted);
+}
+
+static bool memcg_kmem_is_accounted(struct mem_cgroup *memcg)
+{
+ return test_bit(KMEM_ACCOUNTED_THIS, &memcg->kmem_accounted);
+}
+
+static void memcg_kmem_account_parent(struct mem_cgroup *memcg)
+{
+ set_bit(KMEM_ACCOUNTED_PARENT, &memcg->kmem_accounted);
+}
+
+static void memcg_kmem_clear_account_parent(struct mem_cgroup *memcg)
+{
+ clear_bit(KMEM_ACCOUNTED_PARENT, &memcg->kmem_accounted);
+}
+#endif /* CONFIG_MEMCG_KMEM */
+
```

```
 /* Stuffs for move charges at task migration. */
 /*
  * Types of charges to be moved. "move_charge_at_immitgrate" is treated as a
@@ -614,7 +647,7 @@ EXPORT_SYMBOL(__memcg_kmem_free_page);

 static void disarm_kmem_keys(struct mem_cgroup *memcg)
 {
- if (memcg->kmem_accounted)
+ if (test_bit(KMEM_ACCOUNTED_THIS, &memcg->kmem_accounted))
   static_key_slow_dec(&memcg_kmem_enabled_key);
 }
 #else
@@ -4171,17 +4204,54 @@ static ssize_t mem_cgroup_read(struct cgroup *cont, struct cftype
*cft,
 static void memcg_update_kmem_limit(struct mem_cgroup *memcg, u64 val)
 {
 #ifdef CONFIG_MEMCG_KMEM
- /*
-  * Once enabled, can't be disabled. We could in theory disable it if we
-  * haven't yet created any caches, or if we can shrink them all to
-  * death. But it is not worth the trouble.
-  */
+ struct mem_cgroup *iter;
+
   mutex_lock(&set_limit_mutex);
- if (!memcg->kmem_accounted && val != RESOURCE_MAX) {
+ if ((val != RESOURCE_MAX) && memcg_kmem_account(memcg)) {
+
+  /*
+   * Once enabled, can't be disabled. We could in theory disable
+   * it if we haven't yet created any caches, or if we can shrink
+   * them all to death. But it is not worth the trouble
+   */
   static_key_slow_inc(&memcg_kmem_enabled_key);
-  memcg->kmem_accounted = true;
+
+  if (!memcg->use_hierarchy)
+   goto out;
+
+  for_each_mem_cgroup_tree(iter, memcg) {
+   if (iter == memcg)
+    continue;
+   memcg_kmem_account_parent(iter);
+  }
+ } else if ((val == RESOURCE_MAX) && memcg_kmem_clear_account(memcg)) {
+
+  if (!memcg->use_hierarchy)
+   goto out;
```

```
+
+  for_each_mem_cgroup_tree(iter, memcg) {
+   struct mem_cgroup *parent;
+
+   if (iter == memcg)
+    continue;
+   /*
+    * We should only have our parent bit cleared if none
+    * of our parents are accounted. The transversal order
+    * of our iter function forces us to always look at the
+    * parents.
+    */
+   parent = parent_mem_cgroup(iter);
+   for (; parent != memcg; parent = parent_mem_cgroup(iter))
+    if (memcg_kmem_is_accounted(parent))
+     goto noclear;
+   memcg_kmem_clear_account_parent(iter);
+noclear:
+   continue;
+  }
  }
+out:
  mutex_unlock(&set_limit_mutex);
+
 #endif
 }

--
1.7.11.2
```

Subject: [PATCH v2 10/11] memcg: allow a memcg with kmem charges to be destructed.
Posted by Glauber Costa on Thu, 09 Aug 2012 13:01:18 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>

```
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
CC: Michal Hocko <mhocko@suse.cz>
CC: Johannes Weiner <hannes@cmpxchg.org>
CC: Suleiman Souhlal <suleiman@google.com>
---
 mm/memcontrol.c | 17 +++++++++++++++++-
 1 file changed, 16 insertions(+), 1 deletion(-)

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 3d30b79..7c1ea49 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -649,6 +649,11 @@ static void disarm_kmem_keys(struct mem_cgroup *memcg)
 {
  if (test_bit(KMEM_ACCOUNTED_THIS, &memcg->kmem_accounted))
   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)
@@ -4005,6 +4010,7 @@ 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);

@@ -4038,8 +4044,17 @@ 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;
--
1.7.11.2
```

Subject: [PATCH v2 11/11] protect architectures where THREAD_SIZE >= PAGE_SIZE against fork bombs
Posted by Glauber Costa on Thu, 09 Aug 2012 13:01:19 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>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
CC: Michal Hocko <mhocko@suse.cz>
CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Johannes Weiner <hannes@cmpxchg.org>
CC: Suleiman Souhlal <suleiman@google.com>
---
 include/linux/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 dc3ff16..b0b90c3 100644
--- a/kernel/fork.c
+++ b/kernel/fork.c
@@ -142,7 +142,7 @@ 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;
@@ -151,7 +151,7 @@ static struct thread_info *alloc_thread_info_node(struct task_struct *tsk,
 static inline void free_thread_info(struct thread_info *ti)
 {
  arch_release_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;
--
1.7.11.2
```

Subject: Re: [PATCH v2 07/11] mm: Allocate kernel pages to the right memcg
Posted by Greg Thelen on Thu, 09 Aug 2012 16:33:03 GMT
View Forum Message <> Reply to Message

On Thu, Aug 09 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().
>
> Signed-off-by: Glauber Costa <glommer@parallels.com>
> CC: Christoph Lameter <cl@linux.com>
> CC: Pekka Enberg <penberg@cs.helsinki.fi>
> CC: Michal Hocko <mhocko@suse.cz>
> CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
> CC: Johannes Weiner <hannes@cmpxchg.org>
> CC: Suleiman Souhlal <suleiman@google.com>
> ---
> include/linux/gfp.h | 3 +++
> mm/page_alloc.c | 38 ++++++++++++++++++++++++++++++++++++++
> 2 files changed, 41 insertions(+)
>
> diff --git a/include/linux/gfp.h b/include/linux/gfp.h
> index d8eae4d..029570f 100644
> --- a/include/linux/gfp.h
> +++ b/include/linux/gfp.h
> @@ -370,6 +370,9 @@ 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 b956cec..da341dc 100644
> --- a/mm/page_alloc.c
> +++ b/mm/page_alloc.c
> @@ -2532,6 +2532,7 @@ __alloc_pages_nodemask(gfp_t gfp_mask, unsigned int order,
>   struct page *page = NULL;
>   int migratetype = allocflags_to_migratetype(gfp_mask);
>   unsigned int cpuset_mems_cookie;
> + void *handle = NULL;
>
>   gfp_mask &= gfp_allowed_mask;
>
> @@ -2543,6 +2544,13 @@ __alloc_pages_nodemask(gfp_t gfp_mask, unsigned int order,
>   return NULL;
>
>   /*

> + * Will only have any effect when __GFP_KMEMCG is set.
> + * This is verified in the (always inline) callee
> + */
> + if (!memcg_kmem_new_page(gfp_mask, &handle, order))
> + return NULL;
> +
> + /*
>   * Check the zones suitable for the gfp_mask contain at least one
>   * valid zone. It's possible to have an empty zonelist as a result
>   * of GFP_THISNODE and a memoryless node

If memcg_kmem_new_page() succeeds then it may have obtained a memcg
reference with mem_cgroup_get(). I think this reference is leaked when
returning below:

```
 /*
  * Check the zones suitable for the gfp_mask contain at least one
  * valid zone. It's possible to have an empty zonelist as a result
  * of GFP_THISNODE and a memoryless node
  */
 if (unlikely(!zonelist->_zonerefs->zone))
  return NULL;
```

I suspect the easiest fix is to swap the call to memcg_kmem_new_page()
and the (!zonelist->_zonerefs->zone) check.

Subject: Re: [PATCH v2 07/11] mm: Allocate kernel pages to the right memcg
Posted by Glauber Costa on Thu, 09 Aug 2012 16:42:01 GMT
View Forum Message <> Reply to Message

On 08/09/2012 08:33 PM, Greg Thelen wrote:
> On Thu, Aug 09 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().
>>
>> Signed-off-by: Glauber Costa <glommer@parallels.com>
>> CC: Christoph Lameter <cl@linux.com>
>> CC: Pekka Enberg <penberg@cs.helsinki.fi>

>> CC: Michal Hocko <mhocko@suse.cz>
>> CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
>> CC: Johannes Weiner <hannes@cmpxchg.org>
>> CC: Suleiman Souhlal <suleiman@google.com>
>> ---
>> include/linux/gfp.h | 3 +++
>> mm/page_alloc.c | 38 ++++++++++++++++++++++++++++++++++++++
>> 2 files changed, 41 insertions(+)
>>
>> diff --git a/include/linux/gfp.h b/include/linux/gfp.h
>> index d8eae4d..029570f 100644
>> --- a/include/linux/gfp.h
>> +++ b/include/linux/gfp.h
>> @@ -370,6 +370,9 @@ 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 b956cec..da341dc 100644
>> --- a/mm/page_alloc.c
>> +++ b/mm/page_alloc.c
>> @@ -2532,6 +2532,7 @@ __alloc_pages_nodemask(gfp_t gfp_mask, unsigned int order,
>>   struct page *page = NULL;
>>   int migratetype = allocflags_to_migratetype(gfp_mask);
>>   unsigned int cpuset_mems_cookie;
>> + void *handle = NULL;
>>
>>   gfp_mask &= gfp_allowed_mask;
>>
>> @@ -2543,6 +2544,13 @@ __alloc_pages_nodemask(gfp_t gfp_mask, unsigned int order,
>>   return NULL;
>>
>>   /*
>> + * Will only have any effect when __GFP_KMEMCG is set.
>> + * This is verified in the (always inline) callee
>> + */
>> + if (!memcg_kmem_new_page(gfp_mask, &handle, order))
>> + return NULL;
>> +
>> + /*
>>   * Check the zones suitable for the gfp_mask contain at least one
>>   * valid zone. It's possible to have an empty zonelist as a result

>>   * of GFP_THISNODE and a memoryless node
>
> If memcg_kmem_new_page() succeeds then it may have obtained a memcg
> reference with mem_cgroup_get(). I think this reference is leaked when
> returning below:
>
> /*
>   * Check the zones suitable for the gfp_mask contain at least one
>   * valid zone. It's possible to have an empty zonelist as a result
>   * of GFP_THISNODE and a memoryless node
>   */
> if (unlikely(!zonelist->_zonerefs->zone))
>   return NULL;
>
> I suspect the easiest fix is to swap the call to memcg_kmem_new_page()
> and the (!zonelist->_zonerefs->zone) check.
>
You are right, indeed.

Subject: Re: [PATCH v2 01/11] memcg: Make it possible to use the stock for more than one page.
Posted by Michal Hocko on Fri, 10 Aug 2012 15:12:39 GMT
View Forum Message <> Reply to Message

On Thu 09-08-12 17:01:09, Glauber Costa 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.
>
> [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>

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

>
> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
> index 95162c9..bc7bfa7 100644
> --- a/mm/memcontrol.c
> +++ b/mm/memcontrol.c
> @@ -2096,20 +2096,28 @@ 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);
> @@ -2408,7 +2416,7 @@ 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 {
> @@ -2433,7 +2441,7 @@ 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
> --
> 1.7.11.2
>
> --
> To unsubscribe from this list: send the line "unsubscribe cgroups" in
> the body of a message to majordomo@vger.kernel.org
> More majordomo info at http://vger.kernel.org/majordomo-info.html

--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v2 02/11] memcg: Reclaim when more than one page needed.
Posted by Michal Hocko on Fri, 10 Aug 2012 15:42:40 GMT
View Forum Message <> Reply to Message

On Thu 09-08-12 17:01:10, Glauber Costa wrote:
[...]
> @@ -2317,18 +2318,18 @@ static int mem_cgroup_do_charge(struct mem_cgroup *memcg, gfp_t gfp_mask,
>   } else
>   mem_over_limit = mem_cgroup_from_res_counter(fail_res, res);
>   /*
> - * nr_pages can be either a huge page (HPAGE_PMD_NR), a batch
> - * of regular pages (CHARGE_BATCH), or a single regular page (1).
> - *
>   * Never reclaim on behalf of optional batching, retry with a
>   * single page instead.
>   */
> - if (nr_pages == CHARGE_BATCH)
> + if (nr_pages > min_pages)
>   return CHARGE_RETRY;

This is dangerous because THP charges will be retried now while they

previously failed with CHARGE_NOMEM which means that we will keep
attempting potentially endlessly.
Why cannot we simply do if (nr_pages < CHARGE_BATCH) and get rid of the
min_pages altogether?
Also the comment doesn't seem to be valid anymore.

>
> if (!(gfp_mask & __GFP_WAIT))
> return CHARGE_WOULDBLOCK;
>
> + if (gfp_mask & __GFP_NORETRY)
> + return CHARGE_NOMEM;
> +
> ret = mem_cgroup_reclaim(mem_over_limit, gfp_mask, flags);
> if (mem_cgroup_margin(mem_over_limit) >= nr_pages)
> return CHARGE_RETRY;
> @@ -2341,7 +2342,7 @@ static int mem_cgroup_do_charge(struct mem_cgroup *memcg,
gfp_t gfp_mask,
> * unlikely to succeed so close to the limit, and we fall back
> * to regular pages anyway in case of failure.
> */
> - if (nr_pages == 1 && ret)
> + if (nr_pages <= (1 << PAGE_ALLOC_COSTLY_ORDER) && ret)
> return CHARGE_RETRY;
>
> /*
> @@ -2476,7 +2477,8 @@ again:
> nr_oom_retries = MEM_CGROUP_RECLAIM_RETRIES;
> }
>
> - ret = mem_cgroup_do_charge(memcg, gfp_mask, batch, oom_check);
> + ret = mem_cgroup_do_charge(memcg, gfp_mask, batch, nr_pages,
> + oom_check);
> switch (ret) {
> case CHARGE_OK:
> break;
> --
> 1.7.11.2
>
> --
> To unsubscribe from this list: send the line "unsubscribe cgroups" in
> the body of a message to majordomo@vger.kernel.org
> More majordomo info at http://vger.kernel.org/majordomo-info.html

--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v2 03/11] memcg: change defines to an enum
Posted by Michal Hocko on Fri, 10 Aug 2012 15:43:53 GMT
View Forum Message <> Reply to Message

On Thu 09-08-12 17:01:11, 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>
> CC: Michal Hocko <mhocko@suse.cz>
> CC: Johannes Weiner <hannes@cmpxchg.org>
> Acked-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>

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

> ---
> mm/memcontrol.c | 26 ++++++++++++++++++----------
> 1 file changed, 16 insertions(+), 10 deletions(-)
>
> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
> index 2cef99a..b0e29f4 100644
> --- a/mm/memcontrol.c
> +++ b/mm/memcontrol.c
> @@ -393,9 +393,12 @@ 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)
> @@ -3983,7 +3986,8 @@ 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);
> @@ -4019,7 +4023,8 @@ 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;
>
> @@ -4095,7 +4100,8 @@ 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);
> @@ -4423,7 +4429,7 @@ 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;
>
> @@ -4506,7 +4512,7 @@ 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;
>
> @@ -4584,7 +4590,7 @@ 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);
> @@ -4609,7 +4615,7 @@ 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);
>
> --
> 1.7.11.2
>
> --
> To unsubscribe from this list: send the line "unsubscribe cgroups" in
> the body of a message to majordomo@vger.kernel.org
> More majordomo info at http://vger.kernel.org/majordomo-info.html

--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v2 02/11] memcg: Reclaim when more than one page needed.
Posted by KAMEZAWA Hiroyuki on Fri, 10 Aug 2012 16:49:25 GMT
View Forum Message <> Reply to Message

(2012/08/11 0:42), Michal Hocko wrote:
> On Thu 09-08-12 17:01:10, Glauber Costa wrote:
> [...]
>> @@ -2317,18 +2318,18 @@ static int mem_cgroup_do_charge(struct mem_cgroup *memcg,
gfp_t gfp_mask,
>> } else
>> mem_over_limit = mem_cgroup_from_res_counter(fail_res, res);
>> /*
>> - * nr_pages can be either a huge page (HPAGE_PMD_NR), a batch
>> - * of regular pages (CHARGE_BATCH), or a single regular page (1).
>> - *
>> * Never reclaim on behalf of optional batching, retry with a
>> * single page instead.
>> */
>> - if (nr_pages == CHARGE_BATCH)
>> + if (nr_pages > min_pages)
>> return CHARGE_RETRY;
>

> This is dangerous because THP charges will be retried now while they
> previously failed with CHARGE_NOMEM which means that we will keep
> attempting potentially endlessly.

with THP, I thought nr_pages == min_pages, and no retry.

> Why cannot we simply do if (nr_pages < CHARGE_BATCH) and get rid of the
> min_pages altogether?

Hm, I think a slab can be larger than CHARGE_BATCH.

> Also the comment doesn't seem to be valid anymore.
>
I agree it's not clean. Because our assumption on nr_pages are changed,
I think this behavior should not depend on nr_pages value..
Shouldn't we have a flag to indicate "trial-for-batched charge" ?

Thanks,
-Kame

Subject: Re: [PATCH v2 04/11] kmem accounting basic infrastructure
Posted by KAMEZAWA Hiroyuki on Fri, 10 Aug 2012 17:02:32 GMT
View Forum Message <> Reply to Message

(2012/08/09 22:01), Glauber Costa wrote:
> This patch adds the basic infrastructure for the accounting of the slab
> caches. 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.
>
> The code is not enabled until a limit is set. This can be tested by the
> flag "kmem_accounted". This means that after the patch is applied, no
> behavioral changes exists for whoever is still using memcg to control
> their memory usage.
>
> 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)
>
> Signed-off-by: Glauber Costa <glommer@parallels.com>
> CC: Michal Hocko <mhocko@suse.cz>
> CC: Johannes Weiner <hannes@cmpxchg.org>
> Reviewed-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>

Could you add a patch for documentation of this new interface and a text
explaining the behavior of "kmem_accounting" ?

Hm, my concern is the difference of behavior between user page accounting and
kmem accounting...but this is how tcp-accounting is working.

Once you add Documentation, it's okay to add my Ack.

Thanks,
-Kame

> ---
> mm/memcontrol.c | 69
++-
> 1 file changed, 68 insertions(+), 1 deletion(-)
>
> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
> index b0e29f4..54e93de 100644
> --- a/mm/memcontrol.c
> +++ b/mm/memcontrol.c
> @@ -273,6 +273,10 @@ 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.
>   */
> @@ -287,6 +291,7 @@ struct mem_cgroup {
>   * Should the accounting and control be hierarchical, per subtree?
>   */
> bool use_hierarchy;
> + bool kmem_accounted;

```
>
>    bool  oom_lock;
>    atomic_t under_oom;
> @@ -397,6 +402,7 @@ enum res_type {
>    _MEM,
>    _MEMSWAP,
>    _OOM_TYPE,
> + _KMEM,
>   };
>
>   #define MEMFILE_PRIVATE(x, val) ((x) << 16 | (val))
> @@ -1499,6 +1505,10 @@ 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));
>
>    mem_cgroup_print_oom_stat(memcg);
>   }
> @@ -4008,6 +4018,9 @@ 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();
>    }
> @@ -4046,8 +4059,23 @@ 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 = res_counter_set_limit(&memcg->kmem, val);
> +   if (ret)
> +    break;
> +   /*
> +    * Once enabled, can't be disabled. We could in theory
> +    * disable it if we haven't yet created any caches, or
> +    * if we can shrink them all to death.
```

```
> +    *
> +    * But it is not worth the trouble
> +    */
> +   if (!memcg->kmem_accounted && val != RESOURCE_MAX)
> +    memcg->kmem_accounted = true;
> +  } else
> +   return -EINVAL;
>     break;
>    case RES_SOFT_LIMIT:
>     ret = res_counter_memparse_write_strategy(buffer, &val);
> @@ -4113,12 +4141,16 @@ 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;
> @@ -4672,6 +4704,33 @@ static int mem_cgroup_oom_control_write(struct cgroup *cgrp,
>   }
>
>   #ifdef CONFIG_MEMCG_KMEM
> +static struct cftype kmem_cgroup_files[] = {
> + {
> +  .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,
> + },
> + {},
> +};
> +
>   static int memcg_init_kmem(struct mem_cgroup *memcg, struct cgroup_subsys *ss)
>   {
>    return mem_cgroup_sockets_init(memcg, ss);
> @@ -5015,6 +5074,12 @@ mem_cgroup_create(struct cgroup *cont)
>     int cpu;
>     enable_swap_cgroup();
>     parent = NULL;
> +
> +#ifdef CONFIG_MEMCG_KMEM
> +  WARN_ON(cgroup_add_cftypes(&mem_cgroup_subsys,
> +        kmem_cgroup_files));
> +#endif
> +
>     if (mem_cgroup_soft_limit_tree_init())
>      goto free_out;
>     root_mem_cgroup = memcg;
> @@ -5033,6 +5098,7 @@ 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.
> @@ -5043,6 +5109,7 @@ mem_cgroup_create(struct cgroup *cont)
>    } else {
>     res_counter_init(&memcg->res, NULL);
>     res_counter_init(&memcg->memsw, NULL);
> +  res_counter_init(&memcg->kmem, NULL);
>    }
>    memcg->last_scanned_node = MAX_NUMNODES;
>    INIT_LIST_HEAD(&memcg->oom_notify);
>
```

Subject: Re: [PATCH v2 05/11] Add a __GFP_KMEMCG flag

Posted by KAMEZAWA Hiroyuki on Fri, 10 Aug 2012 17:07:31 GMT
View Forum Message <> Reply to Message

(2012/08/09 22:01), 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.
>
> Signed-off-by: Glauber Costa <glommer@parallels.com>
> CC: Christoph Lameter <cl@linux.com>
> CC: Pekka Enberg <penberg@cs.helsinki.fi>
> CC: Michal Hocko <mhocko@suse.cz>
> CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
> CC: Johannes Weiner <hannes@cmpxchg.org>
> CC: Suleiman Souhlal <suleiman@google.com>
> CC: Rik van Riel <riel@redhat.com>
> CC: Mel Gorman <mel@csn.ul.ie>

Okay, so, only memcg-aware allocations are accounted.
It seems a safe way to go.

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

Subject: Re: [PATCH v2 06/11] memcg: kmem controller infrastructure
Posted by KAMEZAWA Hiroyuki on Fri, 10 Aug 2012 17:27:17 GMT
View Forum Message <> Reply to Message

(2012/08/09 22:01), 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 accessors. 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.
>
> [v2: improved comments and standardized function names]
>
> Signed-off-by: Glauber Costa <glommer@parallels.com>
> CC: Christoph Lameter <cl@linux.com>
> CC: Pekka Enberg <penberg@cs.helsinki.fi>
> CC: Michal Hocko <mhocko@suse.cz>
> CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
> CC: Johannes Weiner <hannes@cmpxchg.org>
> ---
> include/linux/memcontrol.h | 79 ++++++++++++++++++++++

```
>   mm/memcontrol.c           | 185 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++
>   2 files changed, 264 insertions(+)
>
> diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
> index 8d9489f..75b247e 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 +400,11 @@ struct sock;
>   #ifdef CONFIG_MEMCG_KMEM
>   void sock_update_memcg(struct sock *sk);
>   void sock_release_memcg(struct sock *sk);
> +
> +#define memcg_kmem_on 1
> +bool __memcg_kmem_new_page(gfp_t gfp, void *handle, int order);
> +void __memcg_kmem_commit_page(struct page *page, void *handle, int order);
> +void __memcg_kmem_free_page(struct page *page, int order);
>   #else
>   static inline void sock_update_memcg(struct sock *sk)
>   {
> @@ -406,6 +412,79 @@ static inline void sock_update_memcg(struct sock *sk)
>   static inline void sock_release_memcg(struct sock *sk)
>   {
>   }
> +
> +#define memcg_kmem_on 0
> +static inline bool
> +__memcg_kmem_new_page(gfp_t gfp, void *handle, int order)
> +{
> + return false;
> +}
> +
> +static inline void  __memcg_kmem_free_page(struct page *page, int order)
> +{
> +}
> +
> +static inline void
> +__memcg_kmem_commit_page(struct page *page, struct mem_cgroup *handle, int order)
> +{
> +}
>   #endif /* CONFIG_MEMCG_KMEM */
```

```
> +
> +/**
> + * memcg_kmem_new_page: verify if a new kmem allocation is allowed.
> + * @gfp: the gfp allocation flags.
> + * @handle: 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_new_page(gfp_t gfp, void *handle, int order)
> +{
> + if (!memcg_kmem_on)
> +  return true;
> + if (!(gfp & __GFP_KMEMCG) || (gfp & __GFP_NOFAIL))
> +  return true;
> + if (in_interrupt() || (!current->mm) || (current->flags & PF_KTHREAD))
> +  return true;
> + return __memcg_kmem_new_page(gfp, handle, order);
> +}
> +
> +/**
> + * memcg_kmem_free_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_free_page(struct page *page, int order)
> +{
> + if (memcg_kmem_on)
> +  __memcg_kmem_free_page(page, order);
> +}
> +
> +/**
> + * memcg_kmem_commit_page: embeds correct memcg in a page
> + * @handle: a pointer to the memcg this was charged against.
> + * @page: pointer to struct page recently allocated
> + * @handle: the memcg structure we charged against
> + * @order: allocation order.
> + *
> + * Needs to be called after memcg_kmem_new_page, 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 @handle to the
> + * corresponding page_cgroup.
> + */
> +static __always_inline void
> +memcg_kmem_commit_page(struct page *page, struct mem_cgroup *handle, int order)
> +{
> + if (memcg_kmem_on)
> + __memcg_kmem_commit_page(page, handle, order);
> +}

Doesn't this 2 functions has no short-cuts ?

if (memcg_kmem_on && handle) ?

Maybe free() needs to access page_cgroup...

> #endif /* _LINUX_MEMCONTROL_H */
>
> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
> index 54e93de..e9824c1 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
> @@ -434,6 +438,9 @@ struct mem_cgroup *mem_cgroup_from_css(struct
cgroup_subsys_state *s)
> #include <net/ip.h>
>
> static bool mem_cgroup_is_root(struct mem_cgroup *memcg);
> +static int memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t gfp, s64 delta);
> +static void memcg_uncharge_kmem(struct mem_cgroup *memcg, s64 delta);
> +
> void sock_update_memcg(struct sock *sk)
> {
> if (mem_cgroup_sockets_enabled) {
> @@ -488,6 +495,118 @@ struct cg_proto *tcp_proto_cgroup(struct mem_cgroup *memcg)
> }

> EXPORT_SYMBOL(tcp_proto_cgroup);
> #endif /* CONFIG_INET */
> +
> +static inline bool memcg_kmem_enabled(struct mem_cgroup *memcg)
> +{
> + return !mem_cgroup_disabled() && !mem_cgroup_is_root(memcg) &&
> + memcg->kmem_accounted;
> +}
> +
> +/*
> + * 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
> + *
> + * Returning true means the allocation is possible.
> + */
> +bool __memcg_kmem_new_page(gfp_t gfp, void *_handle, int order)
> +{
> + struct mem_cgroup *memcg;
> + struct mem_cgroup **handle = (struct mem_cgroup **)_handle;
> + bool ret = true;
> + size_t size;
> + struct task_struct *p;
> +
> + *handle = NULL;
> + rcu_read_lock();
> + p = rcu_dereference(current->mm->owner);
> + memcg = mem_cgroup_from_task(p);
> + if (!memcg_kmem_enabled(memcg))
> + goto out;
> +
> + mem_cgroup_get(memcg);
> +

This mem_cgroup_get() will be a potentioal performance problem.
Don't you have good idea to avoid accessing atomic counter here ?
I think some kind of percpu counter or a feature to disable "move task"
will be a help.

> + size = PAGE_SIZE << order;
> + ret = memcg_charge_kmem(memcg, gfp, size) == 0;
> + if (!ret) {

```
> +  mem_cgroup_put(memcg);
> +  goto out;
> + }
> +
> + *handle = memcg;
> +out:
> + rcu_read_unlock();
> + return ret;
> +}
> +EXPORT_SYMBOL(__memcg_kmem_new_page);
> +
> +void __memcg_kmem_commit_page(struct page *page, void *handle, int order)
> +{
> + struct page_cgroup *pc;
> + struct mem_cgroup *memcg = handle;
> +
> + if (!memcg)
> +  return;
> +
> + WARN_ON(mem_cgroup_is_root(memcg));
> + /* The page allocation must have failed. Revert */
> + if (!page) {
> +  size_t size = PAGE_SIZE << order;
> +
> +  memcg_uncharge_kmem(memcg, size);
> +  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_free_page(struct page *page, int order)
> +{
> + struct mem_cgroup *memcg;
> + size_t size;
> + struct page_cgroup *pc;
> +
> + if (mem_cgroup_disabled())
> +  return;
> +
> + pc = lookup_page_cgroup(page);
> + lock_page_cgroup(pc);
> + memcg = pc->mem_cgroup;
```

> + pc->mem_cgroup = NULL;

shouldn't this happen after checking "Used" bit ?
Ah, BTW, why do you need to clear pc->memcg ?

> + if (!PageCgroupUsed(pc)) {
> + unlock_page_cgroup(pc);
> + return;
> + }
> + ClearPageCgroupUsed(pc);
> + unlock_page_cgroup(pc);
> +
> + /*
> + * Checking if kmem accounted is enabled won't work for uncharge, since
> + * it is possible that the user enabled kmem tracking, allocated, and
> + * then disabled it again.
> + *
> + * We trust if there is a memcg associated with the page, it is a valid
> + * allocation
> + */
> + if (!memcg)
> + return;
> +
> + WARN_ON(mem_cgroup_is_root(memcg));
> + size = (1 << order) << PAGE_SHIFT;
> + memcg_uncharge_kmem(memcg, size);
> + mem_cgroup_put(memcg);

Why do we need ref-counting here ? kmem res_counter cannot work as
reference ?

> +}
> +EXPORT_SYMBOL(__memcg_kmem_free_page);
>   #endif /* CONFIG_MEMCG_KMEM */
>
>   #if defined(CONFIG_INET) && defined(CONFIG_MEMCG_KMEM)
> @@ -5759,3 +5878,69 @@ static int __init enable_swap_account(char *s)
>   __setup("swapaccount=", enable_swap_account);
>
>   #endif
> +
> +#ifdef CONFIG_MEMCG_KMEM
> +int memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t gfp, s64 delta)
> +{

What does 'delta' means ?

```
> + struct res_counter *fail_res;
> + struct mem_cgroup *_memcg;
> + int ret;
> + bool may_oom;
> + bool nofail = false;
> +
> + may_oom = (gfp & __GFP_WAIT) && (gfp & __GFP_FS) &&
> +     !(gfp & __GFP_NORETRY);
> +
> + ret = 0;
> +
> + if (!memcg)
> +  return ret;
> +
> + _memcg = memcg;
> + ret = __mem_cgroup_try_charge(NULL, gfp, delta / PAGE_SIZE,
> +     &_memcg, may_oom);
> +
> + if (ret == -EINTR)  {
> +  nofail = true;
> +  /*
> +   * __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
> +   */
> +  res_counter_charge_nofail(&memcg->res, delta, &fail_res);
> +  if (do_swap_account)
> +   res_counter_charge_nofail(&memcg->memsw, delta,
> +       &fail_res);
> +  ret = 0;
```

Hm, you returns 0 and this charge may never be uncharged....right ?

Thanks,
-Kame

Subject: Re: [PATCH v2 02/11] memcg: Reclaim when more than one page needed.
Posted by Michal Hocko on Fri, 10 Aug 2012 17:28:24 GMT
View Forum Message <> Reply to Message

On Sat 11-08-12 01:49:25, KAMEZAWA Hiroyuki wrote:

> (2012/08/11 0:42), Michal Hocko wrote:
> >On Thu 09-08-12 17:01:10, Glauber Costa wrote:
> >[...]
> >>@@ -2317,18 +2318,18 @@ static int mem_cgroup_do_charge(struct mem_cgroup
*memcg, gfp_t gfp_mask,
> >>   } else
> >>   mem_over_limit = mem_cgroup_from_res_counter(fail_res, res);
> >>   /*
> >>- * nr_pages can be either a huge page (HPAGE_PMD_NR), a batch
> >>- * of regular pages (CHARGE_BATCH), or a single regular page (1).
> >>- *
> >>   * Never reclaim on behalf of optional batching, retry with a
> >>   * single page instead.
> >>   */
> >>- if (nr_pages == CHARGE_BATCH)
> >>+ if (nr_pages > min_pages)
> >>   return CHARGE_RETRY;
> >
> >This is dangerous because THP charges will be retried now while they
> >previously failed with CHARGE_NOMEM which means that we will keep
> >attempting potentially endlessly.
>
> with THP, I thought nr_pages == min_pages, and no retry.

right you are.

> >Why cannot we simply do if (nr_pages < CHARGE_BATCH) and get rid of the
> >min_pages altogether?
>
> Hm, I think a slab can be larger than CHARGE_BATCH.
>
> >Also the comment doesn't seem to be valid anymore.
> >
> I agree it's not clean. Because our assumption on nr_pages are changed,
> I think this behavior should not depend on nr_pages value..
> Shouldn't we have a flag to indicate "trial-for-batched charge" ?

dunno, it would require a new parameter anyway (because abusing gfp
doesn't seem great idea).

>
>
> Thanks,
> -Kame
>
>
>
>

> --
> To unsubscribe from this list: send the line "unsubscribe cgroups" in
> the body of a message to majordomo@vger.kernel.org
> More majordomo info at http://vger.kernel.org/majordomo-info.html

--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v2 02/11] memcg: Reclaim when more than one page needed.
Posted by Michal Hocko on Fri, 10 Aug 2012 17:30:00 GMT
View Forum Message <> Reply to Message

On Thu 09-08-12 17:01:10, Glauber Costa wrote:
[...]
> For now retry up to COSTLY_ORDER (as page_alloc.c does) and make sure
> not to do it if __GFP_NORETRY.

Who is using __GFP_NORETRY for user backed memory (except for hugetlb
which has its own controller)?

--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v2 07/11] mm: Allocate kernel pages to the right memcg
Posted by KAMEZAWA Hiroyuki on Fri, 10 Aug 2012 17:33:06 GMT
View Forum Message <> Reply to Message

(2012/08/09 22:01), 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().
>
> Signed-off-by: Glauber Costa <glommer@parallels.com>
> CC: Christoph Lameter <cl@linux.com>
> CC: Pekka Enberg <penberg@cs.helsinki.fi>

> CC: Michal Hocko <mhocko@suse.cz>
> CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
> CC: Johannes Weiner <hannes@cmpxchg.org>
> CC: Suleiman Souhlal <suleiman@google.com>

Ah, ok. free_accounted_page() seems good.

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

I myself is okay with this. But...

Because you add a new hook to alloc_pages(), please get Ack from Mel
before requesting merge.

Thanks,
-Kame

> ---
> include/linux/gfp.h | 3 +++
> mm/page_alloc.c | 38 ++++++++++++++++++++++++++++++++++++++
> 2 files changed, 41 insertions(+)
>
> diff --git a/include/linux/gfp.h b/include/linux/gfp.h
> index d8eae4d..029570f 100644
> --- a/include/linux/gfp.h
> +++ b/include/linux/gfp.h
> @@ -370,6 +370,9 @@ 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 b956cec..da341dc 100644
> --- a/mm/page_alloc.c
> +++ b/mm/page_alloc.c
> @@ -2532,6 +2532,7 @@ __alloc_pages_nodemask(gfp_t gfp_mask, unsigned int order,
>   struct page *page = NULL;
>   int migratetype = allocflags_to_migratetype(gfp_mask);
>   unsigned int cpuset_mems_cookie;
> + void *handle = NULL;

>
>   gfp_mask &= gfp_allowed_mask;
>
> @@ -2543,6 +2544,13 @@ __alloc_pages_nodemask(gfp_t gfp_mask, unsigned int order,
>   return NULL;
>
>   /*
> + * Will only have any effect when __GFP_KMEMCG is set.
> + * This is verified in the (always inline) callee
> + */
> + if (!memcg_kmem_new_page(gfp_mask, &handle, order))
> + return NULL;
> +
> + /*
>   * Check the zones suitable for the gfp_mask contain at least one
>   * valid zone. It's possible to have an empty zonelist as a result
>   * of GFP_THISNODE and a memoryless node
> @@ -2583,6 +2591,8 @@ out:
>   if (unlikely(!put_mems_allowed(cpuset_mems_cookie) && !page))
>   goto retry_cpuset;
>
> + memcg_kmem_commit_page(page, handle, order);
> +
>   return page;
>   }
>   EXPORT_SYMBOL(__alloc_pages_nodemask);
> @@ -2635,6 +2645,34 @@ 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_free_page(page, order);
> + __free_pages(page, order);
> +}
> +EXPORT_SYMBOL(__free_accounted_pages);
> +

> +void free_accounted_pages(unsigned long addr, unsigned int order)
> +{
> + if (addr != 0) {
> + VM_BUG_ON(!virt_addr_valid((void *)addr));
> + memcg_kmem_free_page(virt_to_page((void *)addr), order);
> + __free_pages(virt_to_page((void *)addr), order);
> + }
> +}
> +EXPORT_SYMBOL(free_accounted_pages);
> +
> static void *make_alloc_exact(unsigned long addr, unsigned order, size_t size)
> {
>   if (addr) {
>

Subject: Re: [PATCH v2 07/11] mm: Allocate kernel pages to the right memcg
Posted by Greg Thelen on Fri, 10 Aug 2012 17:36:58 GMT
View Forum Message <> Reply to Message

On Thu, Aug 09 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().
>
> Signed-off-by: Glauber Costa <glommer@parallels.com>
> CC: Christoph Lameter <cl@linux.com>
> CC: Pekka Enberg <penberg@cs.helsinki.fi>
> CC: Michal Hocko <mhocko@suse.cz>
> CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
> CC: Johannes Weiner <hannes@cmpxchg.org>
> CC: Suleiman Souhlal <suleiman@google.com>
> ---
> include/linux/gfp.h | 3 +++
> mm/page_alloc.c | 38 ++++++++++++++++++++++++++++++++++++++
> 2 files changed, 41 insertions(+)
>
> diff --git a/include/linux/gfp.h b/include/linux/gfp.h
> index d8eae4d..029570f 100644
> --- a/include/linux/gfp.h

> +++ b/include/linux/gfp.h
> @@ -370,6 +370,9 @@ 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 b956cec..da341dc 100644
> --- a/mm/page_alloc.c
> +++ b/mm/page_alloc.c
> @@ -2532,6 +2532,7 @@ __alloc_pages_nodemask(gfp_t gfp_mask, unsigned int order,
>   struct page *page = NULL;
>   int migratetype = allocflags_to_migratetype(gfp_mask);
>   unsigned int cpuset_mems_cookie;
> + void *handle = NULL;
>
>   gfp_mask &= gfp_allowed_mask;
>
> @@ -2543,6 +2544,13 @@ __alloc_pages_nodemask(gfp_t gfp_mask, unsigned int order,
>   return NULL;
>
>   /*
> + * Will only have any effect when __GFP_KMEMCG is set.
> + * This is verified in the (always inline) callee
> + */
> + if (!memcg_kmem_new_page(gfp_mask, &handle, order))
> + return NULL;
> +
> + /*
>   * Check the zones suitable for the gfp_mask contain at least one
>   * valid zone. It's possible to have an empty zonelist as a result
>   * of GFP_THISNODE and a memoryless node
> @@ -2583,6 +2591,8 @@ out:
>   if (unlikely(!put_mems_allowed(cpuset_mems_cookie) && !page))
>   goto retry_cpuset;
>
> + memcg_kmem_commit_page(page, handle, order);
> +
>   return page;
> }
> EXPORT_SYMBOL(__alloc_pages_nodemask);
> @@ -2635,6 +2645,34 @@ 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_free_page(page, order);
> + __free_pages(page, order);
> +}
> +EXPORT_SYMBOL(__free_accounted_pages);
> +
> +void free_accounted_pages(unsigned long addr, unsigned int order)
> +{
> + if (addr != 0) {
> + VM_BUG_ON(!virt_addr_valid((void *)addr));
> + memcg_kmem_free_page(virt_to_page((void *)addr), order);
> + __free_pages(virt_to_page((void *)addr), order);

Nit. Is there any reason not to replace the above two lines with:
__free_accounted_pages(virt_to_page((void *)addr), order);

> + }
> +}
> +EXPORT_SYMBOL(free_accounted_pages);
> +
> static void *make_alloc_exact(unsigned long addr, unsigned order, size_t size)
> {
> if (addr) {

Subject: Re: [PATCH v2 09/11] memcg: propagate kmem limiting information to children
Posted by KAMEZAWA Hiroyuki on Fri, 10 Aug 2012 17:51:06 GMT
View Forum Message <> Reply to Message

(2012/08/09 22:01), Glauber Costa wrote:
> The current memcg slab cache management fails to present satisfatory
> hierarchical behavior in the following scenario:
>

> -> /cgroups/memory/A/B/C
>
> * kmem limit set at A,
> * A and B have no tasks,
> * span a new task in in C.
>
> Because kmem_accounted is a boolean that was not set for C, no
> accounting would be done. This is, however, not what we expect.
>
> The basic idea, is that when a cgroup is limited, we walk the tree
> upwards (something Kame and I already thought about doing for other
> purposes), and make sure that we store the information about the parent
> being limited in kmem_accounted (that is turned into a bitmap: two
> booleans would not be space efficient). The code for that is taken from
> sched/core.c. My reasons for not putting it into a common place is to
> dodge the type issues that would arise from a common implementation
> between memcg and the scheduler - but I think that it should ultimately
> happen, so if you want me to do it now, let me know.
>
> We do the reverse operation when a formerly limited cgroup becomes
> unlimited.
>
> Signed-off-by: Glauber Costa <glommer@parallels.com>
> CC: Christoph Lameter <cl@linux.com>
> CC: Pekka Enberg <penberg@cs.helsinki.fi>
> CC: Michal Hocko <mhocko@suse.cz>
> CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
> CC: Johannes Weiner <hannes@cmpxchg.org>
> CC: Suleiman Souhlal <suleiman@google.com>

> ---
> mm/memcontrol.c | 88 ++------
> 1 file changed, 79 insertions(+), 9 deletions(-)
>
> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
> index 3216292..3d30b79 100644
> --- a/mm/memcontrol.c
> +++ b/mm/memcontrol.c
> @@ -295,7 +295,8 @@ struct mem_cgroup {
> * Should the accounting and control be hierarchical, per subtree?
> */
> bool use_hierarchy;
> - bool kmem_accounted;
> +
> + unsigned long kmem_accounted; /* See KMEM_ACCOUNTED_*, below */
>

>   bool oom_lock;
>   atomic_t under_oom;
> @@ -348,6 +349,38 @@ struct mem_cgroup {
>   #endif
>   };
>
> +enum {
> + KMEM_ACCOUNTED_THIS, /* accounted by this cgroup itself */
> + KMEM_ACCOUNTED_PARENT, /* accounted by any of its parents. */
> +};
> +
> +#ifdef CONFIG_MEMCG_KMEM
> +static bool memcg_kmem_account(struct mem_cgroup *memcg)
> +{
> + return !test_and_set_bit(KMEM_ACCOUNTED_THIS, &memcg->kmem_accounted);
> +}
> +
> +static bool memcg_kmem_clear_account(struct mem_cgroup *memcg)
> +{
> + return test_and_clear_bit(KMEM_ACCOUNTED_THIS, &memcg->kmem_accounted);
> +}
> +
> +static bool memcg_kmem_is_accounted(struct mem_cgroup *memcg)
> +{
> + return test_bit(KMEM_ACCOUNTED_THIS, &memcg->kmem_accounted);
> +}
> +
> +static void memcg_kmem_account_parent(struct mem_cgroup *memcg)
> +{
> + set_bit(KMEM_ACCOUNTED_PARENT, &memcg->kmem_accounted);
> +}
> +
> +static void memcg_kmem_clear_account_parent(struct mem_cgroup *memcg)
> +{
> + clear_bit(KMEM_ACCOUNTED_PARENT, &memcg->kmem_accounted);
> +}
> +#endif /* CONFIG_MEMCG_KMEM */
> +
>   /* Stuffs for move charges at task migration. */
>   /*
>   * Types of charges to be moved. "move_charge_at_immitgrate" is treated as a
> @@ -614,7 +647,7 @@ EXPORT_SYMBOL(__memcg_kmem_free_page);
>
>   static void disarm_kmem_keys(struct mem_cgroup *memcg)
>   {
> - if (memcg->kmem_accounted)
> + if (test_bit(KMEM_ACCOUNTED_THIS, &memcg->kmem_accounted))
>   static_key_slow_dec(&memcg_kmem_enabled_key);

> }
> #else
> @@ -4171,17 +4204,54 @@ static ssize_t mem_cgroup_read(struct cgroup *cont, struct cftype
*cft,
> static void memcg_update_kmem_limit(struct mem_cgroup *memcg, u64 val)
> {
> #ifdef CONFIG_MEMCG_KMEM
> - /*
> - * Once enabled, can't be disabled. We could in theory disable it if we
> - * haven't yet created any caches, or if we can shrink them all to
> - * death. But it is not worth the trouble.
> - */
> + struct mem_cgroup *iter;
> +
>   mutex_lock(&set_limit_mutex);
> - if (!memcg->kmem_accounted && val != RESOURCE_MAX) {
> + if ((val != RESOURCE_MAX) && memcg_kmem_account(memcg)) {
> +
> + /*
> +   * Once enabled, can't be disabled. We could in theory disable
> +   * it if we haven't yet created any caches, or if we can shrink
> +   * them all to death. But it is not worth the trouble
> +   */
>   static_key_slow_inc(&memcg_kmem_enabled_key);
> - memcg->kmem_accounted = true;
> +
> + if (!memcg->use_hierarchy)
> +   goto out;
> +
> + for_each_mem_cgroup_tree(iter, memcg) {
> +   if (iter == memcg)
> +   continue;
> +   memcg_kmem_account_parent(iter);
> + }

Could you add an explanation comment ?

> + } else if ((val == RESOURCE_MAX) && memcg_kmem_clear_account(memcg)) {
> +
> + if (!memcg->use_hierarchy)
> +   goto out;
> +
ditto.

> + for_each_mem_cgroup_tree(iter, memcg) {
> +   struct mem_cgroup *parent;
> +

> + if (iter == memcg)
> + continue;
> + /*
> + * We should only have our parent bit cleared if none
> + * of our parents are accounted. The transversal order
> + * of our iter function forces us to always look at the
> + * parents.
> + */
> + parent = parent_mem_cgroup(iter);
> + for (; parent != memcg; parent = parent_mem_cgroup(iter))
> + if (memcg_kmem_is_accounted(parent))
> + goto noclear;
> + memcg_kmem_clear_account_parent(iter);

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

Subject: Re: [PATCH v2 11/11] protect architectures where THREAD_SIZE >= PAGE_SIZE against fork bombs
Posted by KAMEZAWA Hiroyuki on Fri, 10 Aug 2012 17:54:48 GMT
View Forum Message <> Reply to Message

(2012/08/09 22:01), Glauber Costa wrote:
> 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>
> CC: Christoph Lameter <cl@linux.com>
> CC: Pekka Enberg <penberg@cs.helsinki.fi>
> CC: Michal Hocko <mhocko@suse.cz>

> CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
> CC: Johannes Weiner <hannes@cmpxchg.org>
> CC: Suleiman Souhlal <suleiman@google.com>

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

> ---
> 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 dc3ff16..b0b90c3 100644
> --- a/kernel/fork.c
> +++ b/kernel/fork.c
> @@ -142,7 +142,7 @@ 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;
> @@ -151,7 +151,7 @@ static struct thread_info *alloc_thread_info_node(struct task_struct
*tsk,
> static inline void free_thread_info(struct thread_info *ti)
> {
>   arch_release_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;

>

Subject: Re: [PATCH v2 02/11] memcg: Reclaim when more than one page needed.
Posted by KAMEZAWA Hiroyuki on Fri, 10 Aug 2012 17:56:20 GMT
View Forum Message <> Reply to Message

(2012/08/11 2:28), Michal Hocko wrote:
> On Sat 11-08-12 01:49:25, KAMEZAWA Hiroyuki wrote:
>> (2012/08/11 0:42), Michal Hocko wrote:
>>> On Thu 09-08-12 17:01:10, Glauber Costa wrote:
>>> [...]
>>>> @@ -2317,18 +2318,18 @@ static int mem_cgroup_do_charge(struct mem_cgroup
*memcg, gfp_t gfp_mask,
>>>> } else
>>>> mem_over_limit = mem_cgroup_from_res_counter(fail_res, res);
>>>> /*
>>>> - * nr_pages can be either a huge page (HPAGE_PMD_NR), a batch
>>>> - * of regular pages (CHARGE_BATCH), or a single regular page (1).
>>>> - *
>>>> * Never reclaim on behalf of optional batching, retry with a
>>>> * single page instead.
>>>> */
>>>> - if (nr_pages == CHARGE_BATCH)
>>>> + if (nr_pages > min_pages)
>>>> return CHARGE_RETRY;
>>>
>>> This is dangerous because THP charges will be retried now while they
>>> previously failed with CHARGE_NOMEM which means that we will keep
>>> attempting potentially endlessly.
>>
>> with THP, I thought nr_pages == min_pages, and no retry.
>
> right you are.
>
>>> Why cannot we simply do if (nr_pages < CHARGE_BATCH) and get rid of the
>>> min_pages altogether?
>>
>> Hm, I think a slab can be larger than CHARGE_BATCH.
>>
>>> Also the comment doesn't seem to be valid anymore.
>>>
>> I agree it's not clean. Because our assumption on nr_pages are changed,
>> I think this behavior should not depend on nr_pages value..
>> Shouldn't we have a flag to indicate "trial-for-batched charge" ?
>
> dunno, it would require a new parameter anyway (because abusing gfp

> doesn't seem great idea).
>
ok, agreed.

-Kame

Subject: Re: [PATCH v2 02/11] memcg: Reclaim when more than one page needed.
Posted by Michal Hocko on Fri, 10 Aug 2012 18:52:52 GMT
View Forum Message <> Reply to Message

On Fri 10-08-12 19:30:00, Michal Hocko wrote:
> On Thu 09-08-12 17:01:10, Glauber Costa wrote:
> [...]
> > For now retry up to COSTLY_ORDER (as page_alloc.c does) and make sure
> > not to do it if __GFP_NORETRY.
>
> Who is using __GFP_NORETRY for user backed memory (except for hugetlb
> which has its own controller)?

Bahh, friday brain... GFP_THISNODE used by slab. Sorry for noise.
--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v2 02/11] memcg: Reclaim when more than one page needed.
Posted by Michal Hocko on Fri, 10 Aug 2012 18:54:17 GMT
View Forum Message <> Reply to Message

On Thu 09-08-12 17:01:10, Glauber Costa wrote:
> From: Suleiman Souhlal <ssouhlal@FreeBSD.org>
>
> mem_cgroup_do_charge() was written before kmem accounting, and expects
> three cases: being called for 1 page, being called for a stock of 32
> pages, or being called for a hugepage. If we call for 2 or 3 pages (and
> both the stack and several slabs used in process creation are such, at
> least with the debug options I had), it assumed it's being called for
> stock and just retried without reclaiming.
>
> Fix that by passing down a minsize argument in addition to the csize.
>
> And what to do about that (csize == PAGE_SIZE && ret) retry? If it's
> needed at all (and presumably is since it's there, perhaps to handle
> races), then it should be extended to more than PAGE_SIZE, yet how far?

> And should there be a retry count limit, of what? For now retry up to
> COSTLY_ORDER (as page_alloc.c does) and make sure not to do it if
> __GFP_NORETRY.
>
> [v4: fixed nr pages calculation pointed out by Christoph Lameter]
>
> Signed-off-by: Suleiman Souhlal <suleiman@google.com>
> Signed-off-by: Glauber Costa <glommer@parallels.com>
> Reviewed-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>

I am not happy with the min_pages argument but we can do something more
clever later.

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

> ---
> mm/memcontrol.c | 16 ++++++++-------
> 1 file changed, 9 insertions(+), 7 deletions(-)
>
> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
> index bc7bfa7..2cef99a 100644
> --- a/mm/memcontrol.c
> +++ b/mm/memcontrol.c
> @@ -2294,7 +2294,8 @@ enum {
> };
>
> static int mem_cgroup_do_charge(struct mem_cgroup *memcg, gfp_t gfp_mask,
> -   unsigned int nr_pages, bool oom_check)
> +   unsigned int nr_pages, unsigned int min_pages,
> +   bool oom_check)
> {
>   unsigned long csize = nr_pages * PAGE_SIZE;
>   struct mem_cgroup *mem_over_limit;
> @@ -2317,18 +2318,18 @@ static int mem_cgroup_do_charge(struct mem_cgroup *memcg,
gfp_t gfp_mask,
>   } else
>   mem_over_limit = mem_cgroup_from_res_counter(fail_res, res);
>   /*
> - * nr_pages can be either a huge page (HPAGE_PMD_NR), a batch
> - * of regular pages (CHARGE_BATCH), or a single regular page (1).
> - *
>   * Never reclaim on behalf of optional batching, retry with a
>   * single page instead.
>   */
> - if (nr_pages == CHARGE_BATCH)
> + if (nr_pages > min_pages)
>   return CHARGE_RETRY;
>

> if (!(gfp_mask & __GFP_WAIT))
>   return CHARGE_WOULDBLOCK;
>
> + if (gfp_mask & __GFP_NORETRY)
> + return CHARGE_NOMEM;
> +
> ret = mem_cgroup_reclaim(mem_over_limit, gfp_mask, flags);
> if (mem_cgroup_margin(mem_over_limit) >= nr_pages)
>   return CHARGE_RETRY;
> @@ -2341,7 +2342,7 @@ static int mem_cgroup_do_charge(struct mem_cgroup *memcg,
gfp_t gfp_mask,
>   * unlikely to succeed so close to the limit, and we fall back
>   * to regular pages anyway in case of failure.
>   */
> - if (nr_pages == 1 && ret)
> + if (nr_pages <= (1 << PAGE_ALLOC_COSTLY_ORDER) && ret)
>   return CHARGE_RETRY;
>
> /*
> @@ -2476,7 +2477,8 @@ again:
>   nr_oom_retries = MEM_CGROUP_RECLAIM_RETRIES;
>   }
>
> - ret = mem_cgroup_do_charge(memcg, gfp_mask, batch, oom_check);
> + ret = mem_cgroup_do_charge(memcg, gfp_mask, batch, nr_pages,
> +   oom_check);
>   switch (ret) {
>   case CHARGE_OK:
>   break;
> --
> 1.7.11.2
>
> --
> To unsubscribe from this list: send the line "unsubscribe cgroups" in
> the body of a message to majordomo@vger.kernel.org
> More majordomo info at http://vger.kernel.org/majordomo-info.html

--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v2 06/11] memcg: kmem controller infrastructure
Posted by Greg Thelen on Sat, 11 Aug 2012 05:11:22 GMT
View Forum Message <> Reply to Message

On Thu, Aug 09 2012, 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 accessors. 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.
>
> [v2: improved comments and standardized function names]
>
> Signed-off-by: Glauber Costa <glommer@parallels.com>
> CC: Christoph Lameter <cl@linux.com>
> CC: Pekka Enberg <penberg@cs.helsinki.fi>
> CC: Michal Hocko <mhocko@suse.cz>
> CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
> CC: Johannes Weiner <hannes@cmpxchg.org>
> ---
> include/linux/memcontrol.h | 79 +++++++++++++++++++
> mm/memcontrol.c | 185 +++
> 2 files changed, 264 insertions(+)
>
> diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
> index 8d9489f..75b247e 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 +400,11 @@ struct sock;
> #ifdef CONFIG_MEMCG_KMEM
> void sock_update_memcg(struct sock *sk);
> void sock_release_memcg(struct sock *sk);
> +
> +#define memcg_kmem_on 1
> +bool __memcg_kmem_new_page(gfp_t gfp, void *handle, int order);
> +void __memcg_kmem_commit_page(struct page *page, void *handle, int order);
> +void __memcg_kmem_free_page(struct page *page, int order);
> #else
> static inline void sock_update_memcg(struct sock *sk)
> {
> @@ -406,6 +412,79 @@ static inline void sock_update_memcg(struct sock *sk)
> static inline void sock_release_memcg(struct sock *sk)
> {

```
>  }
> +
> +#define memcg_kmem_on 0
> +static inline bool
> +__memcg_kmem_new_page(gfp_t gfp, void *handle, int order)
> +{
> + return false;
> +}
> +
> +static inline void  __memcg_kmem_free_page(struct page *page, int order)
> +{
> +}
> +
> +static inline void
> +__memcg_kmem_commit_page(struct page *page, struct mem_cgroup *handle, int order)
> +{
> +}
>  #endif /* CONFIG_MEMCG_KMEM */
> +
> +/**
> + * memcg_kmem_new_page: verify if a new kmem allocation is allowed.
> + * @gfp: the gfp allocation flags.
> + * @handle: 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_new_page(gfp_t gfp, void *handle, int order)
> +{
> + if (!memcg_kmem_on)
> +  return true;
> + if (!(gfp & __GFP_KMEMCG) || (gfp & __GFP_NOFAIL))
> +  return true;
> + if (in_interrupt() || (!current->mm) || (current->flags & PF_KTHREAD))
> +  return true;
> + return __memcg_kmem_new_page(gfp, handle, order);
> +}
> +
> +/**
> + * memcg_kmem_free_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_free_page(struct page *page, int order)
> +{
> + if (memcg_kmem_on)
> + __memcg_kmem_free_page(page, order);
> +}
> +
> +/**
> + * memcg_kmem_commit_page: embeds correct memcg in a page
> + * @handle: a pointer to the memcg this was charged against.
> + * @page: pointer to struct page recently allocated
> + * @handle: the memcg structure we charged against
> + * @order: allocation order.
> + *
> + * Needs to be called after memcg_kmem_new_page, 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 @handle to the
> + * corresponding page_cgroup.
> + */
> +static __always_inline void
> +memcg_kmem_commit_page(struct page *page, struct mem_cgroup *handle, int order)
> +{
> + if (memcg_kmem_on)
> + __memcg_kmem_commit_page(page, handle, order);
> +}
> #endif /* _LINUX_MEMCONTROL_H */
>
> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
> index 54e93de..e9824c1 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
> @@ -434,6 +438,9 @@ struct mem_cgroup *mem_cgroup_from_css(struct
cgroup_subsys_state *s)
> #include <net/ip.h>
>

> static bool mem_cgroup_is_root(struct mem_cgroup *memcg);
> +static int memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t gfp, s64 delta);
> +static void memcg_uncharge_kmem(struct mem_cgroup *memcg, s64 delta);
> +
> void sock_update_memcg(struct sock *sk)
> {
>   if (mem_cgroup_sockets_enabled) {
> @@ -488,6 +495,118 @@ struct cg_proto *tcp_proto_cgroup(struct mem_cgroup *memcg)
> }
> EXPORT_SYMBOL(tcp_proto_cgroup);
> #endif /* CONFIG_INET */
> +
> +static inline bool memcg_kmem_enabled(struct mem_cgroup *memcg)
> +{
> + return !mem_cgroup_disabled() && !mem_cgroup_is_root(memcg) &&
> + memcg->kmem_accounted;
> +}
> +
> +/*
> + * 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
> + *
> + * Returning true means the allocation is possible.
> + */
> +bool __memcg_kmem_new_page(gfp_t gfp, void *_handle, int order)
> +{
> + struct mem_cgroup *memcg;
> + struct mem_cgroup **handle = (struct mem_cgroup **)_handle;
> + bool ret = true;
> + size_t size;
> + struct task_struct *p;
> +
> + *handle = NULL;
> + rcu_read_lock();
> + p = rcu_dereference(current->mm->owner);
> + memcg = mem_cgroup_from_task(p);
> + if (!memcg_kmem_enabled(memcg))
> + goto out;
> +
> + mem_cgroup_get(memcg);
> +
> + size = PAGE_SIZE << order;

> + ret = memcg_charge_kmem(memcg, gfp, size) == 0;
> + if (!ret) {
> + mem_cgroup_put(memcg);
> + goto out;
> + }
> +
> + *handle = memcg;
> +out:
> + rcu_read_unlock();
> + return ret;
> +}
> +EXPORT_SYMBOL(__memcg_kmem_new_page);

While running f853d89 from git://github.com/glommer/linux.git , I hit a lockdep issue. To create this I allocated and held reference to some kmem in the context of a kmem limited memcg. Then I moved the allocating process out of memcg and then deleted the memcg. Due to the kmem reference the struct mem_cgroup is still active but invisible in cgroupfs namespace. No problems yet. Then I killed the user process which freed the kmem from the now unlinked memcg. Dropping the kmem caused the memcg ref to hit zero. Then the memcg is deleted but that acquires a non-irqsafe spinlock in softirq which annoys lockdep. I think the lock in question is the mctz below:

```
mem_cgroup_remove_exceeded(struct mem_cgroup *memcg,
    struct mem_cgroup_per_zone *mz,
    struct mem_cgroup_tree_per_zone *mctz)
{
 spin_lock(&mctz->lock);
 __mem_cgroup_remove_exceeded(memcg, mz, mctz);
 spin_unlock(&mctz->lock);
}
```

Perhaps your patches expose this problem by being the first time we call __mem_cgroup_free() from softirq (this is just an educated guess). I'm not sure how this would interact with Ying's soft limit rework: https://lwn.net/Articles/501338/

Here's the dmesg splat.

```
[  335.550398] =================================
[  335.554739] [ INFO: inconsistent lock state ]
[  335.559091] 3.5.0-dbg-DEV #3 Tainted: G        W
[  335.563946] ---------------------------------
[  335.568290] inconsistent {SOFTIRQ-ON-W} -> {IN-SOFTIRQ-W} usage.
[  335.574286] swapper/10/0 [HC0[0]:SC1[1]:HE1:SE0] takes:
[  335.579508]  (&(&rtpz->lock)->rlock){+.?...}, at: [<ffffffff8118216d>]
__mem_cgroup_free+0x8d/0x1b0
```

```
[  335.588525] {SOFTIRQ-ON-W} state was registered at:
[  335.593389]   [<ffffffff810cb073>] __lock_acquire+0x623/0x1a50
[  335.599200]   [<ffffffff810cca55>] lock_acquire+0x95/0x150
[  335.604670]   [<ffffffff81582531>] _raw_spin_lock+0x41/0x50
[  335.610232]   [<ffffffff8118216d>] __mem_cgroup_free+0x8d/0x1b0
[  335.616135]   [<ffffffff811822d5>] mem_cgroup_put+0x45/0x50
[  335.621696]   [<ffffffff81182302>] mem_cgroup_destroy+0x22/0x30
[  335.627592]   [<ffffffff810e093f>] cgroup_diput+0xbf/0x160
[  335.633062]   [<ffffffff811a07ef>] d_delete+0x12f/0x1a0
[  335.638276]   [<ffffffff8119671e>] vfs_rmdir+0x11e/0x140
[  335.643565]   [<ffffffff81199173>] do_rmdir+0x113/0x130
[  335.648773]   [<ffffffff8119a5e6>] sys_rmdir+0x16/0x20
[  335.653900]   [<ffffffff8158c74f>] cstar_dispatch+0x7/0x1f
[  335.659370] irq event stamp: 399732
[  335.662846] hardirqs last  enabled at (399732): [<ffffffff810e8e08>]
res_counter_uncharge_until+0x68/0xa0
[  335.672383] hardirqs last disabled at (399731): [<ffffffff810e8dc8>]
res_counter_uncharge_until+0x28/0xa0
[  335.681916] softirqs last  enabled at (399710): [<ffffffff81085dd3>]
_local_bh_enable+0x13/0x20
[  335.690590] softirqs last disabled at (399711): [<ffffffff8158c48c>] call_softirq+0x1c/0x30
[  335.698914]
[  335.698914] other info that might help us debug this:
[  335.705415]  Possible unsafe locking scenario:
[  335.705415]
[  335.711317]        CPU0
[  335.713757]        ----
[  335.716198]   lock(&(&rtpz->lock)->rlock);
[  335.720282]   <Interrupt>
[  335.722896]     lock(&(&rtpz->lock)->rlock);
[  335.727153]
[  335.727153]  *** DEADLOCK ***
[  335.727153]
[  335.733055] no locks held by swapper/10/0.
[  335.737141]
[  335.737141] stack backtrace:
[  335.741483] Pid: 0, comm: swapper/10 Tainted: G        W    3.5.0-dbg-DEV #3
[  335.748510] Call Trace:
[  335.750952]  <IRQ>  [<ffffffff81579a27>] print_usage_bug+0x1fc/0x20d
[  335.757286]  [<ffffffff81058a9f>] ? save_stack_trace+0x2f/0x50
[  335.763098]  [<ffffffff810ca9ed>] mark_lock+0x29d/0x300
[  335.768309]  [<ffffffff810c9e10>] ? print_irq_inversion_bug.part.36+0x1f0/0x1f0
[  335.775599]  [<ffffffff810caffc>] __lock_acquire+0x5ac/0x1a50
[  335.781323]  [<ffffffff810cad34>] ? __lock_acquire+0x2e4/0x1a50
[  335.787224]  [<ffffffff8118216d>] ? __mem_cgroup_free+0x8d/0x1b0
[  335.793212]  [<ffffffff810cca55>] lock_acquire+0x95/0x150
[  335.798594]  [<ffffffff8118216d>] ? __mem_cgroup_free+0x8d/0x1b0
[  335.804581]  [<ffffffff810e8ddd>] ? res_counter_uncharge_until+0x3d/0xa0
```

[335.811263] [<ffffffff81582531>] _raw_spin_lock+0x41/0x50
[335.816731] [<ffffffff8118216d>] ? __mem_cgroup_free+0x8d/0x1b0
[335.822724] [<ffffffff8118216d>] __mem_cgroup_free+0x8d/0x1b0
[335.828538] [<ffffffff811822d5>] mem_cgroup_put+0x45/0x50
[335.834002] [<ffffffff811828a6>] __memcg_kmem_free_page+0xa6/0x110
[335.840256] [<ffffffff81138109>] free_accounted_pages+0x99/0xa0
[335.846243] [<ffffffff8107b09f>] free_task+0x3f/0x70
[335.851278] [<ffffffff8107b18c>] __put_task_struct+0xbc/0x130
[335.857094] [<ffffffff81081524>] delayed_put_task_struct+0x54/0xd0
[335.863338] [<ffffffff810fd354>] __rcu_process_callbacks+0x1e4/0x490
[335.869757] [<ffffffff810fd62f>] rcu_process_callbacks+0x2f/0x80
[335.875835] [<ffffffff810862f5>] __do_softirq+0xc5/0x270
[335.881218] [<ffffffff810c49b4>] ? clockevents_program_event+0x74/0x100
[335.887895] [<ffffffff810c5d94>] ? tick_program_event+0x24/0x30
[335.893882] [<ffffffff8158c48c>] call_softirq+0x1c/0x30
[335.899179] [<ffffffff8104cefd>] do_softirq+0x8d/0xc0
[335.904301] [<ffffffff810867de>] irq_exit+0xae/0xe0
[335.909251] [<ffffffff8158cc3e>] smp_apic_timer_interrupt+0x6e/0x99
[335.915591] [<ffffffff8158ba9c>] apic_timer_interrupt+0x6c/0x80
[335.921583] <EOI> [<ffffffff810530e7>] ? default_idle+0x67/0x270
[335.927741] [<ffffffff810530e5>] ? default_idle+0x65/0x270

Subject: Re: [PATCH v2 09/11] memcg: propagate kmem limiting information to children
Posted by Glauber Costa on Mon, 13 Aug 2012 08:01:20 GMT
View Forum Message <> Reply to Message

On 08/10/2012 09:51 PM, Kamezawa Hiroyuki wrote:
>> + /*
>> + * Once enabled, can't be disabled. We could in theory disable
>> + * it if we haven't yet created any caches, or if we can shrink
>> + * them all to death. But it is not worth the trouble
>> + */
>> static_key_slow_inc(&memcg_kmem_enabled_key);
>> - memcg->kmem_accounted = true;
>> +
>> + if (!memcg->use_hierarchy)
>> + goto out;
>> +
>> + for_each_mem_cgroup_tree(iter, memcg) {
>> + if (iter == memcg)
>> + continue;
>> + memcg_kmem_account_parent(iter);
>> + }
>
> Could you add an explanation comment ?
>

Of course, Kame.
Thanks

Subject: Re: [PATCH v2 07/11] mm: Allocate kernel pages to the right memcg
Posted by Glauber Costa on Mon, 13 Aug 2012 08:02:16 GMT
View Forum Message <> Reply to Message

On 08/10/2012 09:36 PM, Greg Thelen wrote:
> On Thu, Aug 09 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().
>>
>> Signed-off-by: Glauber Costa <glommer@parallels.com>
>> CC: Christoph Lameter <cl@linux.com>
>> CC: Pekka Enberg <penberg@cs.helsinki.fi>
>> CC: Michal Hocko <mhocko@suse.cz>
>> CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
>> CC: Johannes Weiner <hannes@cmpxchg.org>
>> CC: Suleiman Souhlal <suleiman@google.com>
>> ---
>> include/linux/gfp.h | 3 +++
>> mm/page_alloc.c | 38 ++++++++++++++++++++++++++++++++++++++
>> 2 files changed, 41 insertions(+)
>>
>> diff --git a/include/linux/gfp.h b/include/linux/gfp.h
>> index d8eae4d..029570f 100644
>> --- a/include/linux/gfp.h
>> +++ b/include/linux/gfp.h
>> @@ -370,6 +370,9 @@ 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 b956cec..da341dc 100644
>> --- a/mm/page_alloc.c
>> +++ b/mm/page_alloc.c
>> @@ -2532,6 +2532,7 @@ __alloc_pages_nodemask(gfp_t gfp_mask, unsigned int order,
>> struct page *page = NULL;
>> int migratetype = allocflags_to_migratetype(gfp_mask);
>> unsigned int cpuset_mems_cookie;
>> + void *handle = NULL;
>>
>> gfp_mask &= gfp_allowed_mask;
>>
>> @@ -2543,6 +2544,13 @@ __alloc_pages_nodemask(gfp_t gfp_mask, unsigned int order,
>> return NULL;
>>
>> /*
>> + * Will only have any effect when __GFP_KMEMCG is set.
>> + * This is verified in the (always inline) callee
>> + */
>> + if (!memcg_kmem_new_page(gfp_mask, &handle, order))
>> + return NULL;
>> +
>> + /*
>> * Check the zones suitable for the gfp_mask contain at least one
>> * valid zone. It's possible to have an empty zonelist as a result
>> * of GFP_THISNODE and a memoryless node
>> @@ -2583,6 +2591,8 @@ out:
>> if (unlikely(!put_mems_allowed(cpuset_mems_cookie) && !page))
>> goto retry_cpuset;
>>
>> + memcg_kmem_commit_page(page, handle, order);
>> +
>> return page;
>> }
>> EXPORT_SYMBOL(__alloc_pages_nodemask);
>> @@ -2635,6 +2645,34 @@ 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_free_page(page, order);
>> + __free_pages(page, order);
>> +}
>> +EXPORT_SYMBOL(__free_accounted_pages);
>> +
>> +void free_accounted_pages(unsigned long addr, unsigned int order)
>> +{
>> + if (addr != 0) {
>> + VM_BUG_ON(!virt_addr_valid((void *)addr));
>> + memcg_kmem_free_page(virt_to_page((void *)addr), order);
>> + __free_pages(virt_to_page((void *)addr), order);
>
> Nit. Is there any reason not to replace the above two lines with:
>   __free_accounted_pages(virt_to_page((void *)addr), order);
>
Not any particular reason. If people prefer it this way, I can do that
with no problems.

Subject: Re: [PATCH v2 07/11] mm: Allocate kernel pages to the right memcg
Posted by Glauber Costa on Mon, 13 Aug 2012 08:03:38 GMT
View Forum Message <> Reply to Message

On 08/10/2012 09:33 PM, Kamezawa Hiroyuki wrote:
> (2012/08/09 22:01), 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().
>>
>> Signed-off-by: Glauber Costa <glommer@parallels.com>
>> CC: Christoph Lameter <cl@linux.com>
>> CC: Pekka Enberg <penberg@cs.helsinki.fi>
>> CC: Michal Hocko <mhocko@suse.cz>
>> CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
>> CC: Johannes Weiner <hannes@cmpxchg.org>
>> CC: Suleiman Souhlal <suleiman@google.com>

>
> Ah, ok. free_accounted_page() seems good.
>
> Acked-by: KAMEZAWA Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
>
> I myself is okay with this. But...
>
> Because you add a new hook to alloc_pages(), please get Ack from Mel
> before requesting merge.
>
> Thanks,
> -Kame

Absolutely.

Mel, would you mind taking a look at this series and commenting on this?

Thanks in advance.

Subject: Re: [PATCH v2 02/11] memcg: Reclaim when more than one page needed.
Posted by Glauber Costa on Mon, 13 Aug 2012 08:05:38 GMT
View Forum Message <> Reply to Message

On 08/10/2012 10:54 PM, Michal Hocko wrote:
> On Thu 09-08-12 17:01:10, Glauber Costa wrote:
>> From: Suleiman Souhlal <ssouhlal@FreeBSD.org>
>>
>> mem_cgroup_do_charge() was written before kmem accounting, and expects
>> three cases: being called for 1 page, being called for a stock of 32
>> pages, or being called for a hugepage. If we call for 2 or 3 pages (and
>> both the stack and several slabs used in process creation are such, at
>> least with the debug options I had), it assumed it's being called for
>> stock and just retried without reclaiming.
>>
>> Fix that by passing down a minsize argument in addition to the csize.
>>
>> And what to do about that (csize == PAGE_SIZE && ret) retry? If it's
>> needed at all (and presumably is since it's there, perhaps to handle
>> races), then it should be extended to more than PAGE_SIZE, yet how far?
>> And should there be a retry count limit, of what? For now retry up to
>> COSTLY_ORDER (as page_alloc.c does) and make sure not to do it if
>> __GFP_NORETRY.
>>
>> [v4: fixed nr pages calculation pointed out by Christoph Lameter]
>>
>> Signed-off-by: Suleiman Souhlal <suleiman@google.com>

>> Signed-off-by: Glauber Costa <glommer@parallels.com>
>> Reviewed-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
>
> I am not happy with the min_pages argument but we can do something more
> clever later.
>
> Acked-by: Michal Hocko <mhocko@suse.cz>
>

I am a bit confused here. Does your ack come before or after your other
comments on this patch?

Subject: Re: [PATCH v2 06/11] memcg: kmem controller infrastructure
Posted by Glauber Costa on Mon, 13 Aug 2012 08:07:46 GMT
View Forum Message <> Reply to Message

On 08/11/2012 09:11 AM, Greg Thelen wrote:
> On Thu, Aug 09 2012, 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 accessors. 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.
>>
>> [v2: improved comments and standardized function names]
>>
>> Signed-off-by: Glauber Costa <glommer@parallels.com>
>> CC: Christoph Lameter <cl@linux.com>
>> CC: Pekka Enberg <penberg@cs.helsinki.fi>
>> CC: Michal Hocko <mhocko@suse.cz>
>> CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
>> CC: Johannes Weiner <hannes@cmpxchg.org>
>> ---
>> include/linux/memcontrol.h | 79 ++++++++++++++++++++++
>> mm/memcontrol.c | 185 +++
>> 2 files changed, 264 insertions(+)
>>
>> diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
>> index 8d9489f..75b247e 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 +400,11 @@ struct sock;
>>  #ifdef CONFIG_MEMCG_KMEM
>>  void sock_update_memcg(struct sock *sk);
>>  void sock_release_memcg(struct sock *sk);
>> +
>> +#define memcg_kmem_on 1
>> +bool __memcg_kmem_new_page(gfp_t gfp, void *handle, int order);
>> +void __memcg_kmem_commit_page(struct page *page, void *handle, int order);
>> +void __memcg_kmem_free_page(struct page *page, int order);
>>  #else
>>  static inline void sock_update_memcg(struct sock *sk)
>>  {
>> @@ -406,6 +412,79 @@ static inline void sock_update_memcg(struct sock *sk)
>>  static inline void sock_release_memcg(struct sock *sk)
>>  {
>>  }
>> +
>> +#define memcg_kmem_on 0
>> +static inline bool
>> +__memcg_kmem_new_page(gfp_t gfp, void *handle, int order)
>> +{
>> + return false;
>> +}
>> +
>> +static inline void  __memcg_kmem_free_page(struct page *page, int order)
>> +{
>> +}
>> +
>> +static inline void
>> +__memcg_kmem_commit_page(struct page *page, struct mem_cgroup *handle, int order)
>> +{
>> +}
>>  #endif /* CONFIG_MEMCG_KMEM */
>> +
>> +/**
>> + * memcg_kmem_new_page: verify if a new kmem allocation is allowed.
>> + * @gfp: the gfp allocation flags.
>> + * @handle: 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_new_page(gfp_t gfp, void *handle, int order)
>> +{
>> + if (!memcg_kmem_on)
>> +  return true;
>> + if (!(gfp & __GFP_KMEMCG) || (gfp & __GFP_NOFAIL))
>> +  return true;
>> + if (in_interrupt() || (!current->mm) || (current->flags & PF_KTHREAD))
>> +  return true;
>> + return __memcg_kmem_new_page(gfp, handle, order);
>> +}
>> +
>> +/**
>> + * memcg_kmem_free_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_free_page(struct page *page, int order)
>> +{
>> + if (memcg_kmem_on)
>> +  __memcg_kmem_free_page(page, order);
>> +}
>> +
>> +/**
>> + * memcg_kmem_commit_page: embeds correct memcg in a page
>> + * @handle: a pointer to the memcg this was charged against.
>> + * @page: pointer to struct page recently allocated
>> + * @handle: the memcg structure we charged against
>> + * @order: allocation order.
>> + *
>> + * Needs to be called after memcg_kmem_new_page, 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 @handle to the
>> + * corresponding page_cgroup.
>> + */
>> +static __always_inline void
>> +memcg_kmem_commit_page(struct page *page, struct mem_cgroup *handle, int order)
>> +{
>> + if (memcg_kmem_on)
>> +  __memcg_kmem_commit_page(page, handle, order);
>> +}
>>  #endif /* _LINUX_MEMCONTROL_H */
```

>>
>> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
>> index 54e93de..e9824c1 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
>> @@ -434,6 +438,9 @@ struct mem_cgroup *mem_cgroup_from_css(struct
cgroup_subsys_state *s)
>> #include <net/ip.h>
>>
>> static bool mem_cgroup_is_root(struct mem_cgroup *memcg);
>> +static int memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t gfp, s64 delta);
>> +static void memcg_uncharge_kmem(struct mem_cgroup *memcg, s64 delta);
>> +
>> void sock_update_memcg(struct sock *sk)
>> {
>>   if (mem_cgroup_sockets_enabled) {
>> @@ -488,6 +495,118 @@ struct cg_proto *tcp_proto_cgroup(struct mem_cgroup *memcg)
>> }
>> EXPORT_SYMBOL(tcp_proto_cgroup);
>> #endif /* CONFIG_INET */
>> +
>> +static inline bool memcg_kmem_enabled(struct mem_cgroup *memcg)
>> +{
>> + return !mem_cgroup_disabled() && !mem_cgroup_is_root(memcg) &&
>> + memcg->kmem_accounted;
>> +}
>> +
>> +/*
>> + * 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
>> + *

```
>> + * Returning true means the allocation is possible.
>> + */
>> +bool __memcg_kmem_new_page(gfp_t gfp, void *_handle, int order)
>> +{
>> + struct mem_cgroup *memcg;
>> + struct mem_cgroup **handle = (struct mem_cgroup **)_handle;
>> + bool ret = true;
>> + size_t size;
>> + struct task_struct *p;
>> +
>> + *handle = NULL;
>> + rcu_read_lock();
>> + p = rcu_dereference(current->mm->owner);
>> + memcg = mem_cgroup_from_task(p);
>> + if (!memcg_kmem_enabled(memcg))
>> +  goto out;
>> +
>> + mem_cgroup_get(memcg);
>> +
>> + size = PAGE_SIZE << order;
>> + ret = memcg_charge_kmem(memcg, gfp, size) == 0;
>> + if (!ret) {
>> +  mem_cgroup_put(memcg);
>> +  goto out;
>> + }
>> +
>> + *handle = memcg;
>> +out:
>> + rcu_read_unlock();
>> + return ret;
>> +}
>> +EXPORT_SYMBOL(__memcg_kmem_new_page);
>
> While running f853d89 from git://github.com/glommer/linux.git , I hit a
> lockdep issue.  To create this I allocated and held reference to some
> kmem in the context of a kmem limited memcg.  Then I moved the
> allocating process out of memcg and then deleted the memcg.  Due to the
> kmem reference the struct mem_cgroup is still active but invisible in
> cgroupfs namespace.  No problems yet.  Then I killed the user process
> which freed the kmem from the now unlinked memcg.  Dropping the kmem
> caused the memcg ref to hit zero.  Then the memcg is deleted but that
> acquires a non-irqsafe spinlock in softirq which annoys lockdep.  I
> think the lock in question is the mctz below:
>
>     mem_cgroup_remove_exceeded(struct mem_cgroup *memcg,
>         struct mem_cgroup_per_zone *mz,
>         struct mem_cgroup_tree_per_zone *mctz)
>     {
```

> spin_lock(&mctz->lock);
> __mem_cgroup_remove_exceeded(memcg, mz, mctz);
> spin_unlock(&mctz->lock);
> }
>
> Perhaps your patches expose this problem by being the first time we call
> __mem_cgroup_free() from softirq (this is just an educated guess). I'm
> not sure how this would interact with Ying's soft limit rework:
> https://lwn.net/Articles/501338/
>

Thanks for letting me know, Greg,

I'll try to reproduce this today and see how it goes.

Subject: Re: [PATCH v2 06/11] memcg: kmem controller infrastructure
Posted by Glauber Costa on Mon, 13 Aug 2012 08:28:14 GMT
View Forum Message <> Reply to Message

>> > + * Needs to be called after memcg_kmem_new_page, 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 @handle to the
>> > + * corresponding page_cgroup.
>> > + */
>> > +static __always_inline void
>> > +memcg_kmem_commit_page(struct page *page, struct mem_cgroup *handle, int order)
>> > +{
>> > + if (memcg_kmem_on)
>> > + __memcg_kmem_commit_page(page, handle, order);
>> > +}
> Doesn't this 2 functions has no short-cuts ?

Sorry kame, what exactly do you mean?

> if (memcg_kmem_on && handle) ?
I guess this can be done to avoid a function call.

> Maybe free() needs to access page_cgroup...
>
Can you also be a bit more specific here?

>> > +bool __memcg_kmem_new_page(gfp_t gfp, void *_handle, int order)
>> > +{
>> > + struct mem_cgroup *memcg;
>> > + struct mem_cgroup **handle = (struct mem_cgroup **)_handle;
>> > + bool ret = true;
>> > + size_t size;

>> > + struct task_struct *p;
>> > +
>> > + *handle = NULL;
>> > + rcu_read_lock();
>> > + p = rcu_dereference(current->mm->owner);
>> > + memcg = mem_cgroup_from_task(p);
>> > + if (!memcg_kmem_enabled(memcg))
>> > + goto out;
>> > +
>> > + mem_cgroup_get(memcg);
>> > +
> This mem_cgroup_get() will be a potentioal performance problem.
> Don't you have good idea to avoid accessing atomic counter here ?
> I think some kind of percpu counter or a feature to disable "move task"
> will be a help.

>> > + pc = lookup_page_cgroup(page);
>> > + lock_page_cgroup(pc);
>> > + pc->mem_cgroup = memcg;
>> > + SetPageCgroupUsed(pc);
>> > + unlock_page_cgroup(pc);
>> > +}
>> > +
>> > +void __memcg_kmem_free_page(struct page *page, int order)
>> > +{
>> > + struct mem_cgroup *memcg;
>> > + size_t size;
>> > + struct page_cgroup *pc;
>> > +
>> > + if (mem_cgroup_disabled())
>> > + return;
>> > +
>> > + pc = lookup_page_cgroup(page);
>> > + lock_page_cgroup(pc);
>> > + memcg = pc->mem_cgroup;
>> > + pc->mem_cgroup = NULL;

> shouldn't this happen after checking "Used" bit ?
> Ah, BTW, why do you need to clear pc->memcg ?

As for clearing pc->memcg, I think I'm just being overzealous. I can't foresee any problems due to removing it.

As for the Used bit, what difference does it make when we clear it?

>> > + if (!PageCgroupUsed(pc)) {
>> > + unlock_page_cgroup(pc);
>> > + return;
>> > + }
>> > + ClearPageCgroupUsed(pc);
>> > + unlock_page_cgroup(pc);
>> > +
>> > + /*
>> > + * Checking if kmem accounted is enabled won't work for uncharge, since
>> > + * it is possible that the user enabled kmem tracking, allocated, and
>> > + * then disabled it again.
>> > + *
>> > + * We trust if there is a memcg associated with the page, it is a valid
>> > + * allocation
>> > + */
>> > + if (!memcg)
>> > + return;
>> > +
>> > + WARN_ON(mem_cgroup_is_root(memcg));
>> > + size = (1 << order) << PAGE_SHIFT;
>> > + memcg_uncharge_kmem(memcg, size);
>> > + mem_cgroup_put(memcg);
> Why do we need ref-counting here ? kmem res_counter cannot work as
> reference ?
This is of course the pair of the mem_cgroup_get() you commented on
earlier. If we need one, we need the other. If we don't need one, we
don't need the other =)

The guarantee we're trying to give here is that the memcg structure will
stay around while there are dangling charges to kmem, that we decided
not to move (remember: moving it for the stack is simple, for the slab
is very complicated and ill-defined, and I believe it is better to treat
all kmem equally here)

So maybe we can be clever here, and avoid reference counting at all
times. We call mem_cgroup_get() when the first charge occurs, and then
go for mem_cgroup_put() when our count reaches 0.

What do you think about that?

>> > +#ifdef CONFIG_MEMCG_KMEM
>> > +int memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t gfp, s64 delta)
>> > +{
> What does 'delta' means ?
>
I can change it to something like nr_bytes, more informative.

```
>> > + struct res_counter *fail_res;
>> > + struct mem_cgroup *_memcg;
>> > + int ret;
>> > + bool may_oom;
>> > + bool nofail = false;
>> > +
>> > + may_oom = (gfp & __GFP_WAIT) && (gfp & __GFP_FS) &&
>> > +     !(gfp & __GFP_NORETRY);
>> > +
>> > + ret = 0;
>> > +
>> > + if (!memcg)
>> > +  return ret;
>> > +
>> > + _memcg = memcg;
>> > + ret = __mem_cgroup_try_charge(NULL, gfp, delta / PAGE_SIZE,
>> > +     &_memcg, may_oom);
>> > +
>> > + if (ret == -EINTR)  {
>> > +  nofail = true;
>> > +  /*
>> > +   * __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
>> > +   */
>> > +  res_counter_charge_nofail(&memcg->res, delta, &fail_res);
>> > +  if (do_swap_account)
>> > +   res_counter_charge_nofail(&memcg->memsw, delta,
>> > +       &fail_res);
>> > +  ret = 0;
> Hm, you returns 0 and this charge may never be uncharged....right ?
>
```

Can't see why. By returning 0 we inform our caller that the allocation succeeded. It is up to him to undo it later through a call to uncharge.

Subject: Re: [PATCH v2 04/11] kmem accounting basic infrastructure
Posted by Glauber Costa on Mon, 13 Aug 2012 08:36:51 GMT
View Forum Message <> Reply to Message

On 08/10/2012 09:02 PM, Kamezawa Hiroyuki wrote:
> (2012/08/09 22:01), Glauber Costa wrote:
>> This patch adds the basic infrastructure for the accounting of the slab

>> caches. 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.
>>
>> The code is not enabled until a limit is set. This can be tested by the
>> flag "kmem_accounted". This means that after the patch is applied, no
>> behavioral changes exists for whoever is still using memcg to control
>> their memory usage.
>>
>> 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)
>>
>> Signed-off-by: Glauber Costa <glommer@parallels.com>
>> CC: Michal Hocko <mhocko@suse.cz>
>> CC: Johannes Weiner <hannes@cmpxchg.org>
>> Reviewed-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
>
> Could you add a patch for documentation of this new interface and a text
> explaining the behavior of "kmem_accounting" ?
>
> Hm, my concern is the difference of behavior between user page accounting and
> kmem accounting...but this is how tcp-accounting is working.
>
> Once you add Documentation, it's okay to add my Ack.
>
I plan to add documentation in a separate patch. Due to that, can I add
your ack to this patch here?

Also, I find that the description text in patch0 grew to be quite
informative and complete. I plan to add that to the documentation
if that is ok with you

Subject: Re: [PATCH v2 07/11] mm: Allocate kernel pages to the right memcg

On Mon, Aug 13, 2012 at 12:03:38PM +0400, Glauber Costa wrote:
> On 08/10/2012 09:33 PM, Kamezawa Hiroyuki wrote:
> > (2012/08/09 22:01), 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().
> >>
> >> Signed-off-by: Glauber Costa <glommer@parallels.com>
> >> CC: Christoph Lameter <cl@linux.com>
> >> CC: Pekka Enberg <penberg@cs.helsinki.fi>
> >> CC: Michal Hocko <mhocko@suse.cz>
> >> CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
> >> CC: Johannes Weiner <hannes@cmpxchg.org>
> >> CC: Suleiman Souhlal <suleiman@google.com>
> >
> > Ah, ok. free_accounted_page() seems good.
> >
> > Acked-by: KAMEZAWA Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
> >
> > I myself is okay with this. But...
> >
> > Because you add a new hook to alloc_pages(), please get Ack from Mel
> > before requesting merge.
> >
> > Thanks,
> > -Kame
>
> Absolutely.
>
> Mel, would you mind taking a look at this series and commenting on this?
>

It'll take me a few days but I'll get around to it.

--
Mel Gorman
SUSE Labs

Subject: Re: [PATCH v2 06/11] memcg: kmem controller infrastructure
Posted by Glauber Costa on Mon, 13 Aug 2012 09:59:50 GMT
View Forum Message <> Reply to Message

>
> Here's the dmesg splat.
>

Do you always get this report in the same way?
I managed to get a softirq inconsistency like yours, but the complaint
goes for a different lock.

> [335.550398] ================================
> [335.554739] [INFO: inconsistent lock state]
> [335.559091] 3.5.0-dbg-DEV #3 Tainted: G   W
> [335.563946] --------------------------------
> [335.568290] inconsistent {SOFTIRQ-ON-W} -> {IN-SOFTIRQ-W} usage.
> [335.574286] swapper/10/0 [HC0[0]:SC1[1]:HE1:SE0] takes:
> [335.579508] (&(&rtpz->lock)->rlock){+.?...}, at: [<ffffffff8118216d>]
__mem_cgroup_free+0x8d/0x1b0
> [335.588525] {SOFTIRQ-ON-W} state was registered at:
> [335.593389]   [<ffffffff810cb073>] __lock_acquire+0x623/0x1a50
> [335.599200]   [<ffffffff810cca55>] lock_acquire+0x95/0x150
> [335.604670]   [<ffffffff81582531>] _raw_spin_lock+0x41/0x50
> [335.610232]   [<ffffffff8118216d>] __mem_cgroup_free+0x8d/0x1b0
> [335.616135]   [<ffffffff811822d5>] mem_cgroup_put+0x45/0x50
> [335.621696]   [<ffffffff81182302>] mem_cgroup_destroy+0x22/0x30
> [335.627592]   [<ffffffff810e093f>] cgroup_diput+0xbf/0x160
> [335.633062]   [<ffffffff811a07ef>] d_delete+0x12f/0x1a0
> [335.638276]   [<ffffffff8119671e>] vfs_rmdir+0x11e/0x140
> [335.643565]   [<ffffffff81199173>] do_rmdir+0x113/0x130
> [335.648773]   [<ffffffff8119a5e6>] sys_rmdir+0x16/0x20
> [335.653900]   [<ffffffff8158c74f>] cstar_dispatch+0x7/0x1f
> [335.659370] irq event stamp: 399732
> [335.662846] hardirqs last enabled at (399732): [<ffffffff810e8e08>]
res_counter_uncharge_until+0x68/0xa0
> [335.672383] hardirqs last disabled at (399731): [<ffffffff810e8dc8>]
res_counter_uncharge_until+0x28/0xa0
> [335.681916] softirqs last enabled at (399710): [<ffffffff81085dd3>]
_local_bh_enable+0x13/0x20
> [335.690590] softirqs last disabled at (399711): [<ffffffff8158c48c>] call_softirq+0x1c/0x30
> [335.698914]
> [335.698914] other info that might help us debug this:
> [335.705415] Possible unsafe locking scenario:
> [335.705415]
> [335.711317]   CPU0
> [335.713757]   ----
> [335.716198]   lock(&(&rtpz->lock)->rlock);

```
> [  335.720282]   <Interrupt>
> [  335.722896]     lock(&(&rtpz->lock)->rlock);
> [  335.727153]
> [  335.727153]  *** DEADLOCK ***
> [  335.727153]
> [  335.733055] no locks held by swapper/10/0.
> [  335.737141]
> [  335.737141] stack backtrace:
> [  335.741483] Pid: 0, comm: swapper/10 Tainted: G        W    3.5.0-dbg-DEV #3
> [  335.748510] Call Trace:
> [  335.750952]  <IRQ>  [<ffffffff81579a27>] print_usage_bug+0x1fc/0x20d
> [  335.757286]  [<ffffffff81058a9f>] ? save_stack_trace+0x2f/0x50
> [  335.763098]  [<ffffffff810ca9ed>] mark_lock+0x29d/0x300
> [  335.768309]  [<ffffffff810c9e10>] ? print_irq_inversion_bug.part.36+0x1f0/0x1f0
> [  335.775599]  [<ffffffff810caffc>] __lock_acquire+0x5ac/0x1a50
> [  335.781323]  [<ffffffff810cad34>] ? __lock_acquire+0x2e4/0x1a50
> [  335.787224]  [<ffffffff8118216d>] ? __mem_cgroup_free+0x8d/0x1b0
> [  335.793212]  [<ffffffff810cca55>] lock_acquire+0x95/0x150
> [  335.798594]  [<ffffffff8118216d>] ? __mem_cgroup_free+0x8d/0x1b0
> [  335.804581]  [<ffffffff810e8ddd>] ? res_counter_uncharge_until+0x3d/0xa0
> [  335.811263]  [<ffffffff81582531>] _raw_spin_lock+0x41/0x50
> [  335.816731]  [<ffffffff8118216d>] ? __mem_cgroup_free+0x8d/0x1b0
> [  335.822724]  [<ffffffff8118216d>] __mem_cgroup_free+0x8d/0x1b0
> [  335.828538]  [<ffffffff811822d5>] mem_cgroup_put+0x45/0x50
> [  335.834002]  [<ffffffff811828a6>] __memcg_kmem_free_page+0xa6/0x110
> [  335.840256]  [<ffffffff81138109>] free_accounted_pages+0x99/0xa0
> [  335.846243]  [<ffffffff8107b09f>] free_task+0x3f/0x70
> [  335.851278]  [<ffffffff8107b18c>] __put_task_struct+0xbc/0x130
> [  335.857094]  [<ffffffff81081524>] delayed_put_task_struct+0x54/0xd0
> [  335.863338]  [<ffffffff810fd354>] __rcu_process_callbacks+0x1e4/0x490
> [  335.869757]  [<ffffffff810fd62f>] rcu_process_callbacks+0x2f/0x80
> [  335.875835]  [<ffffffff810862f5>] __do_softirq+0xc5/0x270
> [  335.881218]  [<ffffffff810c49b4>] ? clockevents_program_event+0x74/0x100
> [  335.887895]  [<ffffffff810c5d94>] ? tick_program_event+0x24/0x30
> [  335.893882]  [<ffffffff8158c48c>] call_softirq+0x1c/0x30
> [  335.899179]  [<ffffffff8104cefd>] do_softirq+0x8d/0xc0
> [  335.904301]  [<ffffffff810867de>] irq_exit+0xae/0xe0
> [  335.909251]  [<ffffffff8158cc3e>] smp_apic_timer_interrupt+0x6e/0x99
> [  335.915591]  [<ffffffff8158ba9c>] apic_timer_interrupt+0x6c/0x80
> [  335.921583]  <EOI>  [<ffffffff810530e7>] ? default_idle+0x67/0x270
> [  335.927741]  [<ffffffff810530e5>] ? default_idle+0x65/0x270
>
```

Subject: Re: [PATCH v2 02/11] memcg: Reclaim when more than one page needed.
Posted by Michal Hocko on Mon, 13 Aug 2012 13:10:14 GMT

On Mon 13-08-12 12:05:38, Glauber Costa wrote:
> On 08/10/2012 10:54 PM, Michal Hocko wrote:
> > On Thu 09-08-12 17:01:10, Glauber Costa wrote:
> >> From: Suleiman Souhlal <ssouhlal@FreeBSD.org>
> >>
> >> mem_cgroup_do_charge() was written before kmem accounting, and expects
> >> three cases: being called for 1 page, being called for a stock of 32
> >> pages, or being called for a hugepage. If we call for 2 or 3 pages (and
> >> both the stack and several slabs used in process creation are such, at
> >> least with the debug options I had), it assumed it's being called for
> >> stock and just retried without reclaiming.
> >>
> >> Fix that by passing down a minsize argument in addition to the csize.
> >>
> >> And what to do about that (csize == PAGE_SIZE && ret) retry? If it's
> >> needed at all (and presumably is since it's there, perhaps to handle
> >> races), then it should be extended to more than PAGE_SIZE, yet how far?
> >> And should there be a retry count limit, of what? For now retry up to
> >> COSTLY_ORDER (as page_alloc.c does) and make sure not to do it if
> >> __GFP_NORETRY.
> >>
> >> [v4: fixed nr pages calculation pointed out by Christoph Lameter]
> >>
> >> Signed-off-by: Suleiman Souhlal <suleiman@google.com>
> >> Signed-off-by: Glauber Costa <glommer@parallels.com>
> >> Reviewed-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
> >
> > I am not happy with the min_pages argument but we can do something more
> > clever later.
> >
> > Acked-by: Michal Hocko <mhocko@suse.cz>
> >
>
> I am a bit confused here. Does your ack come before or after your other
> comments on this patch?

Heh, it was hard Friday ;) Yes, it was after the mind fart...
--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v2 06/11] memcg: kmem controller infrastructure
Posted by Greg Thelen on Mon, 13 Aug 2012 21:21:19 GMT

On Mon, Aug 13 2012, Glauber Costa wrote:

>>
>> Here's the dmesg splat.
>>
>
> Do you always get this report in the same way?
> I managed to get a softirq inconsistency like yours, but the complaint
> goes for a different lock.

Yes, I repeatedly get the same dmesg splat below.

Once I your 'execute the whole memcg freeing in rcu callback' patch,
then the warnings are not printed. I'll take a closer look at the patch
soon.

>> [335.550398] =================================
>> [335.554739] [INFO: inconsistent lock state]
>> [335.559091] 3.5.0-dbg-DEV #3 Tainted: G   W
>> [335.563946] ---------------------------------
>> [335.568290] inconsistent {SOFTIRQ-ON-W} -> {IN-SOFTIRQ-W} usage.
>> [335.574286] swapper/10/0 [HC0[0]:SC1[1]:HE1:SE0] takes:
>> [335.579508] (&(&rtpz->lock)->rlock){+.?...}, at: [<ffffffff8118216d>]
__mem_cgroup_free+0x8d/0x1b0
>> [335.588525] {SOFTIRQ-ON-W} state was registered at:
>> [335.593389]   [<ffffffff810cb073>] __lock_acquire+0x623/0x1a50
>> [335.599200]   [<ffffffff810cca55>] lock_acquire+0x95/0x150
>> [335.604670]   [<ffffffff81582531>] _raw_spin_lock+0x41/0x50
>> [335.610232]   [<ffffffff8118216d>] __mem_cgroup_free+0x8d/0x1b0
>> [335.616135]   [<ffffffff811822d5>] mem_cgroup_put+0x45/0x50
>> [335.621696]   [<ffffffff81182302>] mem_cgroup_destroy+0x22/0x30
>> [335.627592]   [<ffffffff810e093f>] cgroup_diput+0xbf/0x160
>> [335.633062]   [<ffffffff811a07ef>] d_delete+0x12f/0x1a0
>> [335.638276]   [<ffffffff8119671e>] vfs_rmdir+0x11e/0x140
>> [335.643565]   [<ffffffff81199173>] do_rmdir+0x113/0x130
>> [335.648773]   [<ffffffff8119a5e6>] sys_rmdir+0x16/0x20
>> [335.653900]   [<ffffffff8158c74f>] cstar_dispatch+0x7/0x1f
>> [335.659370] irq event stamp: 399732
>> [335.662846] hardirqs last enabled at (399732): [<ffffffff810e8e08>]
res_counter_uncharge_until+0x68/0xa0
>> [335.672383] hardirqs last disabled at (399731): [<ffffffff810e8dc8>]
res_counter_uncharge_until+0x28/0xa0
>> [335.681916] softirqs last enabled at (399710): [<ffffffff81085dd3>]
_local_bh_enable+0x13/0x20
>> [335.690590] softirqs last disabled at (399711): [<ffffffff8158c48c>] call_softirq+0x1c/0x30
>> [335.698914]
>> [335.698914] other info that might help us debug this:
>> [335.705415] Possible unsafe locking scenario:

```
>> [  335.705415]
>> [  335.711317]       CPU0
>> [  335.713757]       ----
>> [  335.716198]   lock(&(&rtpz->lock)->rlock);
>> [  335.720282]   <Interrupt>
>> [  335.722896]     lock(&(&rtpz->lock)->rlock);
>> [  335.727153]
>> [  335.727153]  *** DEADLOCK ***
>> [  335.727153]
>> [  335.733055] no locks held by swapper/10/0.
>> [  335.737141]
>> [  335.737141] stack backtrace:
>> [  335.741483] Pid: 0, comm: swapper/10 Tainted: G        W    3.5.0-dbg-DEV #3
>> [  335.748510] Call Trace:
>> [  335.750952]  <IRQ>  [<ffffffff81579a27>] print_usage_bug+0x1fc/0x20d
>> [  335.757286]  [<ffffffff81058a9f>] ? save_stack_trace+0x2f/0x50
>> [  335.763098]  [<ffffffff810ca9ed>] mark_lock+0x29d/0x300
>> [  335.768309]  [<ffffffff810c9e10>] ? print_irq_inversion_bug.part.36+0x1f0/0x1f0
>> [  335.775599]  [<ffffffff810caffc>] __lock_acquire+0x5ac/0x1a50
>> [  335.781323]  [<ffffffff810cad34>] ? __lock_acquire+0x2e4/0x1a50
>> [  335.787224]  [<ffffffff8118216d>] ? __mem_cgroup_free+0x8d/0x1b0
>> [  335.793212]  [<ffffffff810cca55>] lock_acquire+0x95/0x150
>> [  335.798594]  [<ffffffff8118216d>] ? __mem_cgroup_free+0x8d/0x1b0
>> [  335.804581]  [<ffffffff810e8ddd>] ? res_counter_uncharge_until+0x3d/0xa0
>> [  335.811263]  [<ffffffff81582531>] _raw_spin_lock+0x41/0x50
>> [  335.816731]  [<ffffffff8118216d>] ? __mem_cgroup_free+0x8d/0x1b0
>> [  335.822724]  [<ffffffff8118216d>] __mem_cgroup_free+0x8d/0x1b0
>> [  335.828538]  [<ffffffff811822d5>] mem_cgroup_put+0x45/0x50
>> [  335.834002]  [<ffffffff811828a6>] __memcg_kmem_free_page+0xa6/0x110
>> [  335.840256]  [<ffffffff81138109>] free_accounted_pages+0x99/0xa0
>> [  335.846243]  [<ffffffff8107b09f>] free_task+0x3f/0x70
>> [  335.851278]  [<ffffffff8107b18c>] __put_task_struct+0xbc/0x130
>> [  335.857094]  [<ffffffff81081524>] delayed_put_task_struct+0x54/0xd0
>> [  335.863338]  [<ffffffff810fd354>] __rcu_process_callbacks+0x1e4/0x490
>> [  335.869757]  [<ffffffff810fd62f>] rcu_process_callbacks+0x2f/0x80
>> [  335.875835]  [<ffffffff810862f5>] __do_softirq+0xc5/0x270
>> [  335.881218]  [<ffffffff810c49b4>] ? clockevents_program_event+0x74/0x100
>> [  335.887895]  [<ffffffff810c5d94>] ? tick_program_event+0x24/0x30
>> [  335.893882]  [<ffffffff8158c48c>] call_softirq+0x1c/0x30
>> [  335.899179]  [<ffffffff8104cefd>] do_softirq+0x8d/0xc0
>> [  335.904301]  [<ffffffff810867de>] irq_exit+0xae/0xe0
>> [  335.909251]  [<ffffffff8158cc3e>] smp_apic_timer_interrupt+0x6e/0x99
>> [  335.915591]  [<ffffffff8158ba9c>] apic_timer_interrupt+0x6c/0x80
>> [  335.921583]  <EOI>  [<ffffffff810530e7>] ? default_idle+0x67/0x270
>> [  335.927741]  [<ffffffff810530e5>] ? default_idle+0x65/0x270
>>
```

Subject: Re: [PATCH v2 06/11] memcg: kmem controller infrastructure
Posted by Glauber Costa on Tue, 14 Aug 2012 11:00:53 GMT
View Forum Message <> Reply to Message

On 08/10/2012 09:27 PM, Kamezawa Hiroyuki wrote:
>> +bool __memcg_kmem_new_page(gfp_t gfp, void *_handle, int order)
>> > +{
>> > + struct mem_cgroup *memcg;
>> > + struct mem_cgroup **handle = (struct mem_cgroup **)_handle;
>> > + bool ret = true;
>> > + size_t size;
>> > + struct task_struct *p;
>> > +
>> > + *handle = NULL;
>> > + rcu_read_lock();
>> > + p = rcu_dereference(current->mm->owner);
>> > + memcg = mem_cgroup_from_task(p);
>> > + if (!memcg_kmem_enabled(memcg))
>> > + goto out;
>> > +
>> > + mem_cgroup_get(memcg);
>> > +
> This mem_cgroup_get() will be a potentioal performance problem.
> Don't you have good idea to avoid accessing atomic counter here ?
> I think some kind of percpu counter or a feature to disable "move task"
> will be a help.
>
>

I have just sent out a proposal to deal with this. I tried the trick of
marking only the first charge and last uncharge, and it works quite
alright at the cost of a bit test on most calls to memcg_kmem_charge.

Please let me know what you think.

Subject: Re: [PATCH v2 07/11] mm: Allocate kernel pages to the right memcg
Posted by Mel Gorman on Tue, 14 Aug 2012 15:16:16 GMT
View Forum Message <> Reply to Message

On Thu, Aug 09, 2012 at 05:01:15PM +0400, 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,

As you already guessed, doing a page_cgroup in the page allocator free
path would be a no-go.

This is my first time glancing at the series and I'm only paying close
attention to this patch so pardon me if my observations have been made
already.

> 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().
>
> Signed-off-by: Glauber Costa <glommer@parallels.com>
> CC: Christoph Lameter <cl@linux.com>
> CC: Pekka Enberg <penberg@cs.helsinki.fi>
> CC: Michal Hocko <mhocko@suse.cz>
> CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
> CC: Johannes Weiner <hannes@cmpxchg.org>
> CC: Suleiman Souhlal <suleiman@google.com>
> ---
> include/linux/gfp.h | 3 +++
> mm/page_alloc.c | 38 ++++++++++++++++++++++++++++++++++++++
> 2 files changed, 41 insertions(+)
>
> diff --git a/include/linux/gfp.h b/include/linux/gfp.h
> index d8eae4d..029570f 100644
> --- a/include/linux/gfp.h
> +++ b/include/linux/gfp.h
> @@ -370,6 +370,9 @@ 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 b956cec..da341dc 100644
> --- a/mm/page_alloc.c
> +++ b/mm/page_alloc.c
> @@ -2532,6 +2532,7 @@ __alloc_pages_nodemask(gfp_t gfp_mask, unsigned int order,
>   struct page *page = NULL;
>   int migratetype = allocflags_to_migratetype(gfp_mask);
>   unsigned int cpuset_mems_cookie;
> + void *handle = NULL;
>

>   gfp_mask &= gfp_allowed_mask;
>
> @@ -2543,6 +2544,13 @@ __alloc_pages_nodemask(gfp_t gfp_mask, unsigned int order,
>   return NULL;
>
>   /*
> + * Will only have any effect when __GFP_KMEMCG is set.
> + * This is verified in the (always inline) callee
> + */
> + if (!memcg_kmem_new_page(gfp_mask, &handle, order))

memcg_kmem_new_page takes a void * parameter already but here you are
passing in a void **. This probably happens to work because you do this

struct mem_cgroup **handle = (struct mem_cgroup **)_handle;

but that appears to defeat the purpose of having an opaque type as a
"handle". You have to treat it different then passing it into the commit
function because it expects a void *. The motivation for an opaque type
is completely unclear to me and how it is managed with a mix of void *
and void ** is very confusing.

On a similar note I spotted #define memcg_kmem_on 1 . That is also
different just for the sake of it. The convension is to do something
like this

/* This helps us to avoid #ifdef CONFIG_NUMA */
#ifdef CONFIG_NUMA
#define NUMA_BUILD 1
#else
#define NUMA_BUILD 0
#endif

memcg_kmem_on was difficult to guess based on its name. I thought initially
that it would only be active if a memcg existed or at least something like
mem_cgroup_disabled() but it's actually enabled if CONFIG_MEMCG_KMEM is set.

I also find it *very* strange to have a function named as if it is an
allocation-style function when it in fact it's looking up a mem_cgroup
and charging it (and uncharging it in the error path if necessary). If
it was called memcg_kmem_newpage_charge I might have found it a little
better. While I believe you have to take care to avoid confusion with
mem_cgroup_newpage_charge, it would be preferable if the APIs were similar.
memcg is hard enough as it is to understand without having different APIs.

This whole operation also looks very expensive (cgroup lookups, RCU locks
taken etc) but I guess you're willing to take that cost in the same of
isolating containers from each other. However, I strongly suggest that

this overhead is measured in advance. It should not stop the series being
merged as such but it should be understood because if the cost is high
then this feature will be avoided like the plague. I am skeptical that
distributions would enable this by default, at least not without support
for cgroup_disable=kmem

As this thing is called from within the allocator, it's not clear why
__memcg_kmem_new_page is exported. I can't imagine why a module would call
it directly although maybe you cover that somewhere else in the series.

>From the point of view of a hook, that is acceptable but just barely. I have
slammed other hooks because it was possible for a subsystem to override them
meaning the runtime cost could be anything. I did not spot a similar issue
here but if I missed it, it's still unacceptable. At least here the cost
is sortof predictable and only affects memcg because of the __GFP_KMEMCG
check in memcg_kmem_new_page.

> + return NULL;
> +
> + /*
>   * Check the zones suitable for the gfp_mask contain at least one
>   * valid zone. It's possible to have an empty zonelist as a result
>   * of GFP_THISNODE and a memoryless node
> @@ -2583,6 +2591,8 @@ out:
>   if (unlikely(!put_mems_allowed(cpuset_mems_cookie) && !page))
>   goto retry_cpuset;
>
> + memcg_kmem_commit_page(page, handle, order);
> +

As a side note, I'm not keen on how you shortcut these functions. They
are all function calls because memcg_kmem_commit_page() will always call
__memcg_kmem_commit_page() to check the handle once it's compiled in.
The handle==NULL check should have happened in the inline function to save
a few cycles.

This also has the feel that the call of memcg_kmem_commit_page belongs in
prep_new_page() but I recognise that requires passing the opaque handler
around which would be very ugly.

>   return page;
> }
> EXPORT_SYMBOL(__alloc_pages_nodemask);
> @@ -2635,6 +2645,34 @@ 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_free_page(page, order);
> + __free_pages(page, order);
> +}
> +EXPORT_SYMBOL(__free_accounted_pages);

memcg_kmem_new_page makes the following check

+	if (!(gfp & __GFP_KMEMCG) || (gfp & __GFP_NOFAIL))
+		return true;

so if the allocation had __GFP_NOFAIL, it does not get charged but can
still be freed. I didn't check if this is really the case but it looks
very suspicious.

Again, this is a fairly heavy operation.

> +
> +void free_accounted_pages(unsigned long addr, unsigned int order)
> +{
> + if (addr != 0) {
> + VM_BUG_ON(!virt_addr_valid((void *)addr));
> + memcg_kmem_free_page(virt_to_page((void *)addr), order);
> + __free_pages(virt_to_page((void *)addr), order);
> + }
> +}
> +EXPORT_SYMBOL(free_accounted_pages);
> +
> static void *make_alloc_exact(unsigned long addr, unsigned order, size_t size)
> {
>   if (addr) {

--
Mel Gorman
SUSE Labs

Subject: Re: [PATCH v2 04/11] kmem accounting basic infrastructure
Posted by Michal Hocko on Tue, 14 Aug 2012 16:21:55 GMT
View Forum Message <> Reply to Message

On Thu 09-08-12 17:01:12, Glauber Costa wrote:
> This patch adds the basic infrastructure for the accounting of the slab
> caches. 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.
>
> The code is not enabled until a limit is set. This can be tested by the
> flag "kmem_accounted". This means that after the patch is applied, no
> behavioral changes exists for whoever is still using memcg to control
> their memory usage.
>
> 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.

Well, it contributes to the user limit so it is not unlimited. It just
falls under a different limit and it tends to contribute less. This can
be quite confusing. I am still not sure whether we should mix the two
things together. If somebody wants to limit the kernel memory he has to
touch the other limit anyway. Do you have a strong reason to mix the
user and kernel counters?
My impression was that kernel allocation should simply fail while user
allocations might reclaim as well. Why should we reclaim just because of
the kernel allocation (which is unreclaimable from hard limit reclaim
point of view)?
I also think that the whole thing would get much simpler if those two
are split. Anyway if this is really a must then this should be
documented here.

One nit bellow.

> 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)
>
> Signed-off-by: Glauber Costa <glommer@parallels.com>
> CC: Michal Hocko <mhocko@suse.cz>

> CC: Johannes Weiner <hannes@cmpxchg.org>
> Reviewed-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
> ---
> mm/memcontrol.c | 69
++-
> 1 file changed, 68 insertions(+), 1 deletion(-)
>
> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
> index b0e29f4..54e93de 100644
> --- a/mm/memcontrol.c
> +++ b/mm/memcontrol.c
[...]
> @@ -4046,8 +4059,23 @@ 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 = res_counter_set_limit(&memcg->kmem, val);
> +   if (ret)
> +   break;
> +   /*
> +   * Once enabled, can't be disabled. We could in theory
> +   * disable it if we haven't yet created any caches, or
> +   * if we can shrink them all to death.
> +   *
> +   * But it is not worth the trouble
> +   */
> +   if (!memcg->kmem_accounted && val != RESOURCE_MAX)
> +   memcg->kmem_accounted = true;
> + } else
> +   return -EINVAL;
>   break;

This doesn't check for the hierachy so kmem_accounted might not be in
sync with it's parents. mem_cgroup_create (below) needs to copy
kmem_accounted down from the parent and the above needs to check if this
is a similar dance like mem_cgroup_oom_control_write.

[...]

> @@ -5033,6 +5098,7 @@ 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.
> @@ -5043,6 +5109,7 @@ mem_cgroup_create(struct cgroup *cont)
>   } else {
>   res_counter_init(&memcg->res, NULL);
>   res_counter_init(&memcg->memsw, NULL);
> + res_counter_init(&memcg->kmem, NULL);
>   }
>   memcg->last_scanned_node = MAX_NUMNODES;
>   INIT_LIST_HEAD(&memcg->oom_notify);
> --
> 1.7.11.2
>
> --
> To unsubscribe from this list: send the line "unsubscribe cgroups" in
> the body of a message to majordomo@vger.kernel.org
> More majordomo info at http://vger.kernel.org/majordomo-info.html

--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v2 06/11] memcg: kmem controller infrastructure
Posted by Michal Hocko on Tue, 14 Aug 2012 17:25:40 GMT
View Forum Message <> Reply to Message

On Thu 09-08-12 17:01:14, 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 accessors. 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.
>
> [v2: improved comments and standardized function names]
>
> Signed-off-by: Glauber Costa <glommer@parallels.com>
> CC: Christoph Lameter <cl@linux.com>
> CC: Pekka Enberg <penberg@cs.helsinki.fi>
> CC: Michal Hocko <mhocko@suse.cz>
> CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
> CC: Johannes Weiner <hannes@cmpxchg.org>
> ---
> include/linux/memcontrol.h | 79 ++++++++++++++++++++++
> mm/memcontrol.c | 185 ++

> 2 files changed, 264 insertions(+)
>
> diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
> index 8d9489f..75b247e 100644
> --- a/include/linux/memcontrol.h
> +++ b/include/linux/memcontrol.h
[...]
> +/**
> + * memcg_kmem_new_page: verify if a new kmem allocation is allowed.
> + * @gfp: the gfp allocation flags.
> + * @handle: 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_new_page(gfp_t gfp, void *handle, int order)
> +{
> + if (!memcg_kmem_on)
> + return true;
> + if (!(gfp & __GFP_KMEMCG) || (gfp & __GFP_NOFAIL))

OK, I see the point behind __GFP_NOFAIL but it would deserve a comment
or a mention in the changelog.

[...]
> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
> index 54e93de..e9824c1 100644
> --- a/mm/memcontrol.c
> +++ b/mm/memcontrol.c
[...]
> +EXPORT_SYMBOL(__memcg_kmem_new_page);

Why is this exported?

> +
> +void __memcg_kmem_commit_page(struct page *page, void *handle, int order)
> +{
> + struct page_cgroup *pc;
> + struct mem_cgroup *memcg = handle;
> +
> + if (!memcg)
> + return;
> +

> + WARN_ON(mem_cgroup_is_root(memcg));
> + /* The page allocation must have failed. Revert */
> + if (!page) {
> + size_t size = PAGE_SIZE << order;
> +
> + memcg_uncharge_kmem(memcg, size);
> + mem_cgroup_put(memcg);
> + return;
> + }
> +
> + pc = lookup_page_cgroup(page);
> + lock_page_cgroup(pc);
> + pc->mem_cgroup = memcg;
> + SetPageCgroupUsed(pc);

Don't we need a write barrier before assigning memcg? Same as
__mem_cgroup_commit_charge. This tests the Used bit always from within
lock_page_cgroup so it should be safe but I am not 100% sure about the
rest of the code.

[...]
> +EXPORT_SYMBOL(__memcg_kmem_free_page);

Why is the symbol exported?

> #endif /* CONFIG_MEMCG_KMEM */
>
> #if defined(CONFIG_INET) && defined(CONFIG_MEMCG_KMEM)
> @@ -5759,3 +5878,69 @@ static int __init enable_swap_account(char *s)
> __setup("swapaccount=", enable_swap_account);
>
> #endif
> +
> +#ifdef CONFIG_MEMCG_KMEM
> +int memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t gfp, s64 delta)
> +{
> + struct res_counter *fail_res;
> + struct mem_cgroup *_memcg;
> + int ret;
> + bool may_oom;
> + bool nofail = false;
> +
> + may_oom = (gfp & __GFP_WAIT) && (gfp & __GFP_FS) &&
> +   !(gfp & __GFP_NORETRY);

This deserves a comment.

> +

> + ret = 0;
> +
> + if (!memcg)
> + return ret;
> +
> + _memcg = memcg;
> + ret = __mem_cgroup_try_charge(NULL, gfp, delta / PAGE_SIZE,
> +   &_memcg, may_oom);

This is really dangerous because atomic allocation which seem to be possible could result in deadlocks because of the reclaim. Also, as I have mentioned in the other email in this thread. Why should we reclaim just because of kernel allocation when we are not reclaiming any of it because shrink_slab is ignored in the memcg reclaim.

> +
> + if (ret == -EINTR) {
> + nofail = true;
> + /*
> +   * __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
> +   */
> + res_counter_charge_nofail(&memcg->res, delta, &fail_res);
> + if (do_swap_account)
> +   res_counter_charge_nofail(&memcg->memsw, delta,
> +   &fail_res);

Hmmm, this is kind of ugly but I guess unvoidable with the current implementation. Oh well...

> + ret = 0;
> + } else if (ret == -ENOMEM)
> + return ret;
> +
> + if (nofail)
> + res_counter_charge_nofail(&memcg->kmem, delta, &fail_res);
> + else
> + ret = res_counter_charge(&memcg->kmem, delta, &fail_res);
> +
> + if (ret) {
> + res_counter_uncharge(&memcg->res, delta);
> + if (do_swap_account)
> +   res_counter_uncharge(&memcg->memsw, delta);

> + }
> +
> + return ret;
> +}
> +
[...]

--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v2 06/11] memcg: kmem controller infrastructure
Posted by Greg Thelen on Tue, 14 Aug 2012 18:58:10 GMT
View Forum Message <> Reply to Message

On Mon, Aug 13 2012, Glauber Costa wrote:

>>> > + WARN_ON(mem_cgroup_is_root(memcg));
>>> > + size = (1 << order) << PAGE_SHIFT;
>>> > + memcg_uncharge_kmem(memcg, size);
>>> > + mem_cgroup_put(memcg);
>> Why do we need ref-counting here ? kmem res_counter cannot work as
>> reference ?
> This is of course the pair of the mem_cgroup_get() you commented on
> earlier. If we need one, we need the other. If we don't need one, we
> don't need the other =)
>
> The guarantee we're trying to give here is that the memcg structure will
> stay around while there are dangling charges to kmem, that we decided
> not to move (remember: moving it for the stack is simple, for the slab
> is very complicated and ill-defined, and I believe it is better to treat
> all kmem equally here)

By keeping memcg structures hanging around until the last referring kmem page is uncharged do such zombie memcg each consume a css_id and thus put pressure on the 64k css_id space? I imagine in pathological cases this would prevent creation of new cgroups until these zombies are dereferenced.

Is there any way to see how much kmem such zombie memcg are consuming? I think we could find these with for_each_mem_cgroup_tree(root_mem_cgroup). Basically, I'm wanting to know where kernel memory has been allocated. For live memcg, an admin can cat memory.kmem.usage_in_bytes. But for zombie memcg, I'm not sure how to get this info. It looks like the root_mem_cgroup memory.kmem.usage_in_bytes is not hierarchically charged.

Subject: Re: [PATCH v2 07/11] mm: Allocate kernel pages to the right memcg
Posted by Glauber Costa on Wed, 15 Aug 2012 09:08:08 GMT
View Forum Message <> Reply to Message

On 08/14/2012 07:16 PM, Mel Gorman wrote:
> On Thu, Aug 09, 2012 at 05:01:15PM +0400, 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,
>
> As you already guessed, doing a page_cgroup in the page allocator free
> path would be a no-go.

Specifically yes, but in general, you will be able to observe that I am
taking all the possible measures to make sure existing paths are
disturbed as little as possible.

Thanks for your review here

>>
>> diff --git a/mm/page_alloc.c b/mm/page_alloc.c
>> index b956cec..da341dc 100644
>> --- a/mm/page_alloc.c
>> +++ b/mm/page_alloc.c
>> @@ -2532,6 +2532,7 @@ __alloc_pages_nodemask(gfp_t gfp_mask, unsigned int order,
>> struct page *page = NULL;
>> int migratetype = allocflags_to_migratetype(gfp_mask);
>> unsigned int cpuset_mems_cookie;
>> + void *handle = NULL;
>>
>> gfp_mask &= gfp_allowed_mask;
>>
>> @@ -2543,6 +2544,13 @@ __alloc_pages_nodemask(gfp_t gfp_mask, unsigned int order,
>> return NULL;
>>
>> /*
>> + * Will only have any effect when __GFP_KMEMCG is set.
>> + * This is verified in the (always inline) callee
>> + */
>> + if (!memcg_kmem_new_page(gfp_mask, &handle, order))
>
> memcg_kmem_new_page takes a void * parameter already but here you are
> passing in a void **. This probably happens to work because you do this
>
> struct mem_cgroup **handle = (struct mem_cgroup **)_handle;
>

> but that appears to defeat the purpose of having an opaque type as a
> "handle". You have to treat it different then passing it into the commit
> function because it expects a void *. The motivation for an opaque type
> is completely unclear to me and how it is managed with a mix of void *
> and void ** is very confusing.

okay.

The opaque exists because I am doing speculative charging. I believe it
to be a better and less complicated approach then letting a page appear
and then charging it. Besides being consistent with the rest of memcg,
it won't create unnecessary disturbance in the page allocator
when the allocation is to fail.

Now, tasks can move between memcgs, so we can't rely on grabbing it from
current in commit_page, so we pass it around as a handle. Also, even if
the task could not move, we already got it once from the task, and that
is not for free. Better save it.

Aside from the handle needed, the cost is more or less the same compared
to doing it in one pass. All we do by using speculative charging is to
split the cost in two, and doing it from two places.
We'd have to charge + update page_cgroup anyway.

As for the type, do you think using struct mem_cgroup would be less
confusing?

> On a similar note I spotted #define memcg_kmem_on 1 . That is also
> different just for the sake of it. The convension is to do something
> like this
>
> /* This helps us to avoid #ifdef CONFIG_NUMA */
> #ifdef CONFIG_NUMA
> #define NUMA_BUILD 1
> #else
> #define NUMA_BUILD 0
> #endif

For simple defines, yes. But a later patch will turn this into a static
branch test. memcg_kmem_on will be always 0 when compile-disabled, but
when enable will expand to static_branch(&...).

> memcg_kmem_on was difficult to guess based on its name. I thought initially
> that it would only be active if a memcg existed or at least something like
> mem_cgroup_disabled() but it's actually enabled if CONFIG_MEMCG_KMEM is set.

For now. And I thought that adding the static branch in this patch would
only confuse matters. The placeholder is there, but it is later patched
to the final thing.

With that explained, if you want me to change it to something else, I
can do it. Should I ?

> I also find it *very* strange to have a function named as if it is an
> allocation-style function when it in fact it's looking up a mem_cgroup
> and charging it (and uncharging it in the error path if necessary). If
> it was called memcg_kmem_newpage_charge I might have found it a little
> better.

I don't feel strongly about names in general. I can change it.
Will update to memcg_kmem_newpage_charge() and memcg_kmem_page_uncharge().

> This whole operation also looks very expensive (cgroup lookups, RCU locks
> taken etc) but I guess you're willing to take that cost in the same of
> isolating containers from each other. However, I strongly suggest that
> this overhead is measured in advance. It should not stop the series being
> merged as such but it should be understood because if the cost is high
> then this feature will be avoided like the plague. I am skeptical that
> distributions would enable this by default, at least not without support
> for cgroup_disable=kmem

Enabling this feature will bring you nothing, therefore, no (or little)
overhead. Nothing of this will be patched in until the first memcg gets
kmem limited. The mere fact of moving tasks to memcgs won't trigger any
of this.

I haven't measured this series in particular, but I did measure the slab
series (which builds ontop of this). I found the per-allocation cost to
be in the order of 2-3 % for tasks living in limited memcgs, and
hard to observe when living in the root memcg (compared of course to the
case of a task running on root memcg without those patches)

I also believe the folks from google also measured this. They may be
able to spit out numbers grabbed from a system bigger than mine =p

> As this thing is called from within the allocator, it's not clear why
> __memcg_kmem_new_page is exported. I can't imagine why a module would call
> it directly although maybe you cover that somewhere else in the series.

Okay, more people commented on this, so let me clarify: They shouldn't
be. They were initially exported when this was about the slab only,
because they could be called from inlined functions from the allocators.
Now that the charge/uncharge was moved to the page allocator - which

already allowed me the big benefit of separating this in two pieces,
none of this needs to be exported.

Sorry for not noticing this myself, but thanks for the eyes =)

> From the point of view of a hook, that is acceptable but just barely. I have
> slammed other hooks because it was possible for a subsystem to override them
> meaning the runtime cost could be anything. I did not spot a similar issue
> here but if I missed it, it's still unacceptable. At least here the cost
> is sortof predictable and only affects memcg because of the __GFP_KMEMCG
> check in memcg_kmem_new_page.

Yes, that is the idea. And I don't think anyone should override those,
so I don't see them as hooks in this sense.

>> + return NULL;
>> +
>> + /*
>>   * Check the zones suitable for the gfp_mask contain at least one
>>   * valid zone. It's possible to have an empty zonelist as a result
>>   * of GFP_THISNODE and a memoryless node
>> @@ -2583,6 +2591,8 @@ out:
>>   if (unlikely(!put_mems_allowed(cpuset_mems_cookie) && !page))
>>   goto retry_cpuset;
>>
>> + memcg_kmem_commit_page(page, handle, order);
>> +
>
> As a side note, I'm not keen on how you shortcut these functions. They
> are all function calls because memcg_kmem_commit_page() will always call
> __memcg_kmem_commit_page() to check the handle once it's compiled in.
> The handle==NULL check should have happened in the inline function to save
> a few cycles.
>

It is already happening on my updated series after a comment from Kame
pointed this out.

> This also has the feel that the call of memcg_kmem_commit_page belongs in
> prep_new_page() but I recognise that requires passing the opaque handler
> around which would be very ugly.

Indeed, and that is the reason why I kept everything local.

>>   return page;

>

> memcg_kmem_new_page makes the following check
>
> +	if (!(gfp & __GFP_KMEMCG) || (gfp & __GFP_NOFAIL))
> +		return true;
>
> so if the allocation had __GFP_NOFAIL, it does not get charged but can
> still be freed. I didn't check if this is really the case but it looks
> very suspicious.

No, it can't be freed (uncharged), because in that case, we won't fill
in the memcg information in page cgroup.

>
> Again, this is a fairly heavy operation.

Mel, once I address all the issues you pointed out here, do you think
this would be in an acceptable state for merging? Do you still have any
fundamental opposition to this?

thanks again

Subject: Re: [PATCH v2 06/11] memcg: kmem controller infrastructure
Posted by Glauber Costa on Wed, 15 Aug 2012 09:18:14 GMT
View Forum Message <> Reply to Message

On 08/14/2012 10:58 PM, Greg Thelen wrote:
> On Mon, Aug 13 2012, Glauber Costa wrote:
>
>>>>> + WARN_ON(mem_cgroup_is_root(memcg));
>>>>> + size = (1 << order) << PAGE_SHIFT;
>>>>> + memcg_uncharge_kmem(memcg, size);
>>>>> + mem_cgroup_put(memcg);
>>> Why do we need ref-counting here ? kmem res_counter cannot work as
>>> reference ?
>> This is of course the pair of the mem_cgroup_get() you commented on
>> earlier. If we need one, we need the other. If we don't need one, we
>> don't need the other =)
>>
>> The guarantee we're trying to give here is that the memcg structure will
>> stay around while there are dangling charges to kmem, that we decided
>> not to move (remember: moving it for the stack is simple, for the slab
>> is very complicated and ill-defined, and I believe it is better to treat
>> all kmem equally here)
>
> By keeping memcg structures hanging around until the last referring kmem
> page is uncharged do such zombie memcg each consume a css_id and thus

> put pressure on the 64k css_id space? I imagine in pathological cases
> this would prevent creation of new cgroups until these zombies are
> dereferenced.

Yes, but although this patch makes it more likely, it doesn't introduce
that. If the tasks, for instance, grab a reference to the cgroup dentry
in the filesystem (like their CWD, etc), they will also keep the cgroup
around.

> Is there any way to see how much kmem such zombie memcg are consuming?
> I think we could find these with
> for_each_mem_cgroup_tree(root_mem_cgroup).

Yes, just need an interface for that. But I think it is something that
can be addressed orthogonaly to this work, in a separate patch, not as
some fundamental limitation.

> Basically, I'm wanting to
> know where kernel memory has been allocated. For live memcg, an admin
> can cat memory.kmem.usage_in_bytes. But for zombie memcg, I'm not sure
> how to get this info. It looks like the root_mem_cgroup
> memory.kmem.usage_in_bytes is not hierarchically charged.
>

Not sure what you mean by not being hierarchically charged. It should
be, when use_hierarchy = 1. As a matter of fact, I just tested it, and I
do see kmem being charged all the way to the root cgroup when hierarchy
is used. (we just can't limit it there)

Subject: Re: [PATCH v2 07/11] mm: Allocate kernel pages to the right memcg
Posted by Michal Hocko on Wed, 15 Aug 2012 09:24:05 GMT
View Forum Message <> Reply to Message

On Thu 09-08-12 17:01:15, Glauber Costa wrote:
[...]
> diff --git a/mm/page_alloc.c b/mm/page_alloc.c
> index b956cec..da341dc 100644
> --- a/mm/page_alloc.c
> +++ b/mm/page_alloc.c
> @@ -2532,6 +2532,7 @@ __alloc_pages_nodemask(gfp_t gfp_mask, unsigned int order,
>   struct page *page = NULL;
>   int migratetype = allocflags_to_migratetype(gfp_mask);
>   unsigned int cpuset_mems_cookie;
> + void *handle = NULL;
>
>   gfp_mask &= gfp_allowed_mask;

>
> @@ -2543,6 +2544,13 @@ __alloc_pages_nodemask(gfp_t gfp_mask, unsigned int order,
> return NULL;
>
> /*
> + * Will only have any effect when __GFP_KMEMCG is set.
> + * This is verified in the (always inline) callee
> + */
> + if (!memcg_kmem_new_page(gfp_mask, &handle, order))
> + return NULL;

When the previous patch introduced this function I thought the handle
obfuscantion is to prevent from spreading struct mem_cgroup inside the
page allocator but memcg_kmem_commit_page uses the type directly. So why
that obfuscation? Even handle as a name sounds unnecessarily confusing.
I would go with struct mem_cgroup **memcgp or even return the pointer on
success or NULL otherwise.

[...]
> +EXPORT_SYMBOL(__free_accounted_pages);

Why exported?

Btw. this is called from call_rcu context but it itself calls call_rcu
down the chain in mem_cgroup_put. Is it safe?

[...]
> +EXPORT_SYMBOL(free_accounted_pages);

here again
--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v2 04/11] kmem accounting basic infrastructure
Posted by Glauber Costa on Wed, 15 Aug 2012 09:33:55 GMT
View Forum Message <> Reply to Message

>> 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.
>
> Well, it contributes to the user limit so it is not unlimited. It just
> falls under a different limit and it tends to contribute less.

You are right, but this is just wording. I will update it, but what I
really mean here is that an independent limit is no imposed on kmem.

> This can
> be quite confusing. I am still not sure whether we should mix the two
> things together. If somebody wants to limit the kernel memory he has to
> touch the other limit anyway. Do you have a strong reason to mix the
> user and kernel counters?

This is funny, because the first opposition I found to this work was
"Why would anyone want to limit it separately?" =p

It seems that a quite common use case is to have a container with a
unified view of "memory" that it can use the way he likes, be it with
kernel memory, or user memory. I believe those people would be happy to
just silently account kernel memory to user memory, or at the most have
a switch to enable it.

What gets clear from this back and forth, is that there are people
interested in both use cases.

> My impression was that kernel allocation should simply fail while user
> allocations might reclaim as well. Why should we reclaim just because of
> the kernel allocation (which is unreclaimable from hard limit reclaim
> point of view)?

That is not what the kernel does, in general. We assume that if he wants
that memory and we can serve it, we should. Also, not all kernel memory
is unreclaimable. We can shrink the slabs, for instance. Ying Han
claims she has patches for that already...

> I also think that the whole thing would get much simpler if those two
> are split. Anyway if this is really a must then this should be
> documented here.

Well, documentation can't hurt.

>
> This doesn't check for the hierachy so kmem_accounted might not be in
> sync with it's parents. mem_cgroup_create (below) needs to copy
> kmem_accounted down from the parent and the above needs to check if this
> is a similar dance like mem_cgroup_oom_control_write.
>

I don't see why we have to.

I believe in a A/B/C hierarchy, C should be perfectly able to set a
different limit than its parents. Note that this is not a boolean.

Also, right now, C can become completely unlimited (by not setting a
limited) and this is, indeed, not the desired behavior.

A later patch will change kmem_accounted to a bitfield, and we'll use
one of the bits to signal that we should account kmem because our parent
is limited.

Subject: Re: [PATCH v2 06/11] memcg: kmem controller infrastructure
Posted by Glauber Costa on Wed, 15 Aug 2012 09:42:24 GMT
View Forum Message <> Reply to Message

>> + * memcg_kmem_new_page: verify if a new kmem allocation is allowed.
>> + * @gfp: the gfp allocation flags.
>> + * @handle: 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_new_page(gfp_t gfp, void *handle, int order)
>> +{
>> + if (!memcg_kmem_on)
>> + return true;
>> + if (!(gfp & __GFP_KMEMCG) || (gfp & __GFP_NOFAIL))
>
> OK, I see the point behind __GFP_NOFAIL but it would deserve a comment
> or a mention in the changelog.

documentation can't hurt!

Just added.

> [...]
>> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
>> index 54e93de..e9824c1 100644
>> --- a/mm/memcontrol.c
>> +++ b/mm/memcontrol.c
> [...]
>> +EXPORT_SYMBOL(__memcg_kmem_new_page);
>
> Why is this exported?
>

It shouldn't be. Removed.

>> +
>> +void __memcg_kmem_commit_page(struct page *page, void *handle, int order)
>> +{
>> + struct page_cgroup *pc;
>> + struct mem_cgroup *memcg = handle;
>> +
>> + if (!memcg)
>> + return;
>> +
>> + WARN_ON(mem_cgroup_is_root(memcg));
>> + /* The page allocation must have failed. Revert */
>> + if (!page) {
>> + size_t size = PAGE_SIZE << order;
>> +
>> + memcg_uncharge_kmem(memcg, size);
>> + mem_cgroup_put(memcg);
>> + return;
>> + }
>> +
>> + pc = lookup_page_cgroup(page);
>> + lock_page_cgroup(pc);
>> + pc->mem_cgroup = memcg;
>> + SetPageCgroupUsed(pc);
>
> Don't we need a write barrier before assigning memcg? Same as
> __mem_cgroup_commit_charge. This tests the Used bit always from within
> lock_page_cgroup so it should be safe but I am not 100% sure about the
> rest of the code.
>
Well, I don't see the reason, precisely because we'll always grab it
from within the locked region. That should ensure all the necessary
serialization.

>> +#ifdef CONFIG_MEMCG_KMEM
>> +int memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t gfp, s64 delta)
>> +{
>> + struct res_counter *fail_res;
>> + struct mem_cgroup *_memcg;
>> + int ret;
>> + bool may_oom;
>> + bool nofail = false;
>> +
>> + may_oom = (gfp & __GFP_WAIT) && (gfp & __GFP_FS) &&
>> +   !(gfp & __GFP_NORETRY);
>

> This deserves a comment.
>
can't hurt!! =)

>> +
>> + ret = 0;
>> +
>> + if (!memcg)
>> + return ret;
>> +
>> + _memcg = memcg;
>> + ret = __mem_cgroup_try_charge(NULL, gfp, delta / PAGE_SIZE,
>> +   &_memcg, may_oom);
>
> This is really dangerous because atomic allocation which seem to be
> possible could result in deadlocks because of the reclaim.

Can you elaborate on how this would happen?

> Also, as I
> have mentioned in the other email in this thread. Why should we reclaim
> just because of kernel allocation when we are not reclaiming any of it
> because shrink_slab is ignored in the memcg reclaim.

Don't get too distracted by the fact that shrink_slab is ignored. It is
temporary, and while this being ignored now leads to suboptimal
behavior, it will 1st, only affect its users, and 2nd, not be disastrous.

I see it this as more or less on pair with the soft limit reclaim
problem we had. It is not ideal, but it already provided functionality

>> +
>> + if (ret == -EINTR) {
>> + nofail = true;
>> + /*
>> +   * __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
>> +   */
>> + res_counter_charge_nofail(&memcg->res, delta, &fail_res);
>> + if (do_swap_account)
>> +   res_counter_charge_nofail(&memcg->memsw, delta,
>> +   &fail_res);

>
> Hmmm, this is kind of ugly but I guess unvoidable with the current
> implementation. Oh well...
>

Oh well...

Subject: Re: [PATCH v2 06/11] memcg: kmem controller infrastructure
Posted by Glauber Costa on Wed, 15 Aug 2012 10:44:15 GMT
View Forum Message <> Reply to Message

On 08/15/2012 01:42 PM, Glauber Costa wrote:
>> Also, as I
>> > have mentioned in the other email in this thread. Why should we reclaim
>> > just because of kernel allocation when we are not reclaiming any of it
>> > because shrink_slab is ignored in the memcg reclaim.
>
> Don't get too distracted by the fact that shrink_slab is ignored. It is
> temporary, and while this being ignored now leads to suboptimal
> behavior, it will 1st, only affect its users, and 2nd, not be disastrous.
>
> I see it this as more or less on pair with the soft limit reclaim
> problem we had. It is not ideal, but it already provided functionality
>

Okay, I sent the e-mail before finishing it... duh

What I meant in this last sentence, is that the situation while the memcg-aware shrinkers doesn't land in the kernel is more or less the same (obviously not exactly) as with the soft reclaim work. It is an evolutionary approach that provides some functionality that is not yet perfect but already solves lots of problems for people willing to live with its temporary drawbacks.

Subject: Re: [PATCH v2 04/11] kmem accounting basic infrastructure
Posted by James Bottomley on Wed, 15 Aug 2012 11:12:23 GMT
View Forum Message <> Reply to Message

On Wed, 2012-08-15 at 13:33 +0400, Glauber Costa wrote:
> > This can
> > be quite confusing. I am still not sure whether we should mix the two
> > things together. If somebody wants to limit the kernel memory he has to
> > touch the other limit anyway. Do you have a strong reason to mix the
> > user and kernel counters?
>

> This is funny, because the first opposition I found to this work was
> "Why would anyone want to limit it separately?" =p
>
> It seems that a quite common use case is to have a container with a
> unified view of "memory" that it can use the way he likes, be it with
> kernel memory, or user memory. I believe those people would be happy to
> just silently account kernel memory to user memory, or at the most have
> a switch to enable it.
>
> What gets clear from this back and forth, is that there are people
> interested in both use cases.

Haven't we already had this discussion during the Prague get together?
We discussed the use cases and finally agreed to separate accounting for
k and then k+u mem because that satisfies both the Google and Parallels
cases. No-one was overjoyed by k and k+u but no-one had a better
suggestion ... is there a better way of doing this that everyone can
agree to?

We do need to get this nailed down because it's the foundation of the
patch series.

James

Subject: Re: [PATCH v2 04/11] kmem accounting basic infrastructure
Posted by Michal Hocko on Wed, 15 Aug 2012 12:39:31 GMT
View Forum Message <> Reply to Message

On Wed 15-08-12 13:33:55, Glauber Costa wrote:
[...]
> > This can
> > be quite confusing. I am still not sure whether we should mix the two
> > things together. If somebody wants to limit the kernel memory he has to
> > touch the other limit anyway. Do you have a strong reason to mix the
> > user and kernel counters?
>
> This is funny, because the first opposition I found to this work was
> "Why would anyone want to limit it separately?" =p
>
> It seems that a quite common use case is to have a container with a
> unified view of "memory" that it can use the way he likes, be it with
> kernel memory, or user memory. I believe those people would be happy to
> just silently account kernel memory to user memory, or at the most have
> a switch to enable it.
>
> What gets clear from this back and forth, is that there are people
> interested in both use cases.

I am still not 100% sure myself. It is just clear that the reclaim would
need some work in order to do accounting like this.

> > My impression was that kernel allocation should simply fail while user
> > allocations might reclaim as well. Why should we reclaim just because of
> > the kernel allocation (which is unreclaimable from hard limit reclaim
> > point of view)?
>
> That is not what the kernel does, in general. We assume that if he wants
> that memory and we can serve it, we should. Also, not all kernel memory
> is unreclaimable. We can shrink the slabs, for instance. Ying Han
> claims she has patches for that already...

Are those patches somewhere around?

[...]
> > This doesn't check for the hierachy so kmem_accounted might not be in
> > sync with it's parents. mem_cgroup_create (below) needs to copy
> > kmem_accounted down from the parent and the above needs to check if this
> > is a similar dance like mem_cgroup_oom_control_write.
> >
>
> I don't see why we have to.
>
> I believe in a A/B/C hierarchy, C should be perfectly able to set a
> different limit than its parents. Note that this is not a boolean.

Ohh, I wasn't clear enough. I am not against setting the _limit_ I just
meant that the kmem_accounted should be consistent within the hierarchy.

--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v2 04/11] kmem accounting basic infrastructure
Posted by Glauber Costa on Wed, 15 Aug 2012 12:53:40 GMT
View Forum Message <> Reply to Message

On 08/15/2012 04:39 PM, Michal Hocko wrote:
> On Wed 15-08-12 13:33:55, Glauber Costa wrote:
> [...]
>>> This can
>>> be quite confusing. I am still not sure whether we should mix the two
>>> things together. If somebody wants to limit the kernel memory he has to
>>> touch the other limit anyway. Do you have a strong reason to mix the
>>> user and kernel counters?

>>
>> This is funny, because the first opposition I found to this work was
>> "Why would anyone want to limit it separately?" =p
>>
>> It seems that a quite common use case is to have a container with a
>> unified view of "memory" that it can use the way he likes, be it with
>> kernel memory, or user memory. I believe those people would be happy to
>> just silently account kernel memory to user memory, or at the most have
>> a switch to enable it.
>>
>> What gets clear from this back and forth, is that there are people
>> interested in both use cases.
>
> I am still not 100% sure myself. It is just clear that the reclaim would
> need some work in order to do accounting like this.
>

Note: Besides what I've already said, right *now* in this series we are
accounting just stack. So reclaimable vs not-reclaimable doesn't even
get to play. It is used while the tasks are running, it gets freed after
the tasks exited.

I do agree we need to look to the whole picture, and reclaiming will be
hard to get right.

This is actually why we're addressing them separately: because they are
a hard problem on their own, and the current status of accounting
already solve real life problems for many, though not for all.

>>> My impression was that kernel allocation should simply fail while user
>>> allocations might reclaim as well. Why should we reclaim just because of
>>> the kernel allocation (which is unreclaimable from hard limit reclaim
>>> point of view)?
>>
>> That is not what the kernel does, in general. We assume that if he wants
>> that memory and we can serve it, we should. Also, not all kernel memory
>> is unreclaimable. We can shrink the slabs, for instance. Ying Han
>> claims she has patches for that already...
>
> Are those patches somewhere around?
>

Ying Han ?

> [...]
>>> This doesn't check for the hierachy so kmem_accounted might not be in
>>> sync with it's parents. mem_cgroup_create (below) needs to copy
>>> kmem_accounted down from the parent and the above needs to check if this

>>> is a similar dance like mem_cgroup_oom_control_write.
>>>
>>
>> I don't see why we have to.
>>
>> I believe in a A/B/C hierarchy, C should be perfectly able to set a
>> different limit than its parents. Note that this is not a boolean.
>
> Ohh, I wasn't clear enough. I am not against setting the _limit_ I just
> meant that the kmem_accounted should be consistent within the hierarchy.
>

If a parent of yours is accounted, you get accounted as well. This is
not the state in this patch, but gets added later. Isn't this enough ?

Subject: Re: [PATCH v2 04/11] kmem accounting basic infrastructure
Posted by Michal Hocko on Wed, 15 Aug 2012 12:55:55 GMT
View Forum Message <> Reply to Message

On Wed 15-08-12 12:12:23, James Bottomley wrote:
> On Wed, 2012-08-15 at 13:33 +0400, Glauber Costa wrote:
> > > This can
> > > be quite confusing. I am still not sure whether we should mix the two
> > > things together. If somebody wants to limit the kernel memory he has to
> > > touch the other limit anyway. Do you have a strong reason to mix the
> > > user and kernel counters?
> >
> > This is funny, because the first opposition I found to this work was
> > "Why would anyone want to limit it separately?" =p
> >
> > It seems that a quite common use case is to have a container with a
> > unified view of "memory" that it can use the way he likes, be it with
> > kernel memory, or user memory. I believe those people would be happy to
> > just silently account kernel memory to user memory, or at the most have
> > a switch to enable it.
> >
> > What gets clear from this back and forth, is that there are people
> > interested in both use cases.
>
> Haven't we already had this discussion during the Prague get together?
> We discussed the use cases and finally agreed to separate accounting for
> k and then k+u mem because that satisfies both the Google and Parallels
> cases. No-one was overjoyed by k and k+u but no-one had a better
> suggestion ... is there a better way of doing this that everyone can
> agree to?
> We do need to get this nailed down because it's the foundation of the
> patch series.

There is a slot in MM/memcg minisum at KS so we have a slot to discuss
this.

>
> James
>
>
> --
> To unsubscribe from this list: send the line "unsubscribe cgroups" in
> the body of a message to majordomo@vger.kernel.org
> More majordomo info at http://vger.kernel.org/majordomo-info.html

--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v2 04/11] kmem accounting basic infrastructure
Posted by Michal Hocko on Wed, 15 Aug 2012 13:02:28 GMT
View Forum Message <> Reply to Message

On Wed 15-08-12 16:53:40, Glauber Costa wrote:
[...]
> >>> This doesn't check for the hierachy so kmem_accounted might not be in
> >>> sync with it's parents. mem_cgroup_create (below) needs to copy
> >>> kmem_accounted down from the parent and the above needs to check if this
> >>> is a similar dance like mem_cgroup_oom_control_write.
> >>>
> >>
> >> I don't see why we have to.
> >>
> >> I believe in a A/B/C hierarchy, C should be perfectly able to set a
> >> different limit than its parents. Note that this is not a boolean.
> >
> > Ohh, I wasn't clear enough. I am not against setting the _limit_ I just
> > meant that the kmem_accounted should be consistent within the hierarchy.
> >
>
> If a parent of yours is accounted, you get accounted as well. This is
> not the state in this patch, but gets added later. Isn't this enough ?

But if the parent is not accounted, you can set the children to be
accounted, right? Or maybe this is changed later in the series? I didn't
get to the end yet.
--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v2 04/11] kmem accounting basic infrastructure
Posted by Glauber Costa on Wed, 15 Aug 2012 13:04:31 GMT
View Forum Message <> Reply to Message

On 08/15/2012 05:02 PM, Michal Hocko wrote:
> On Wed 15-08-12 16:53:40, Glauber Costa wrote:
> [...]
>>>>> This doesn't check for the hierachy so kmem_accounted might not be in
>>>>> sync with it's parents. mem_cgroup_create (below) needs to copy
>>>>> kmem_accounted down from the parent and the above needs to check if this
>>>>> is a similar dance like mem_cgroup_oom_control_write.
>>>>>
>>>>
>>>> I don't see why we have to.
>>>>
>>>> I believe in a A/B/C hierarchy, C should be perfectly able to set a
>>>> different limit than its parents. Note that this is not a boolean.
>>>
>>> Ohh, I wasn't clear enough. I am not against setting the _limit_ I just
>>> meant that the kmem_accounted should be consistent within the hierarchy.
>>>
>>
>> If a parent of yours is accounted, you get accounted as well. This is
>> not the state in this patch, but gets added later. Isn't this enough ?
>
> But if the parent is not accounted, you can set the children to be
> accounted, right? Or maybe this is changed later in the series? I didn't
> get to the end yet.
>

Yes, you can. Do you see any problem with that?

Subject: Re: [PATCH v2 06/11] memcg: kmem controller infrastructure
Posted by Michal Hocko on Wed, 15 Aug 2012 13:09:52 GMT
View Forum Message <> Reply to Message

On Wed 15-08-12 13:42:24, Glauber Costa wrote:
[...]
> >> +
> >> + ret = 0;
> >> +
> >> + if (!memcg)
> >> + return ret;
> >> +
> >> + _memcg = memcg;
> >> + ret = __mem_cgroup_try_charge(NULL, gfp, delta / PAGE_SIZE,
> >> + &_memcg, may_oom);

> >
> > This is really dangerous because atomic allocation which seem to be
> > possible could result in deadlocks because of the reclaim.
>
> Can you elaborate on how this would happen?

Say you have an atomic allocation and we hit the limit so we get either
to reclaim which can sleep or to oom which can sleep as well (depending
on the oom_control).

> > Also, as I have mentioned in the other email in this thread. Why
> > should we reclaim just because of kernel allocation when we are not
> > reclaiming any of it because shrink_slab is ignored in the memcg
> > reclaim.
>
> Don't get too distracted by the fact that shrink_slab is ignored. It is
> temporary, and while this being ignored now leads to suboptimal
> behavior, it will 1st, only affect its users, and 2nd, not be disastrous.

It's not just about shrink_slab it is also about triggering memcg-oom
which doesn't consider kmem accounted memory so the wrong tasks could
be killed. It is true that the impact is packed inside the group
(hierarchy) so you are right it won't be disastrous.
--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v2 07/11] mm: Allocate kernel pages to the right memcg
Posted by Mel Gorman on Wed, 15 Aug 2012 13:22:44 GMT
View Forum Message <> Reply to Message

On Wed, Aug 15, 2012 at 01:08:08PM +0400, Glauber Costa wrote:
> On 08/14/2012 07:16 PM, Mel Gorman wrote:
> > On Thu, Aug 09, 2012 at 05:01:15PM +0400, 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,
> >
> > As you already guessed, doing a page_cgroup in the page allocator free
> > path would be a no-go.
>
> Specifically yes, but in general, you will be able to observe that I am
> taking all the possible measures to make sure existing paths are
> disturbed as little as possible.

>
> Thanks for your review here
>
> >>
> >> diff --git a/mm/page_alloc.c b/mm/page_alloc.c
> >> index b956cec..da341dc 100644
> >> --- a/mm/page_alloc.c
> >> +++ b/mm/page_alloc.c
> >> @@ -2532,6 +2532,7 @@ __alloc_pages_nodemask(gfp_t gfp_mask, unsigned int order,
> >> struct page *page = NULL;
> >> int migratetype = allocflags_to_migratetype(gfp_mask);
> >> unsigned int cpuset_mems_cookie;
> >> + void *handle = NULL;
> >>
> >> gfp_mask &= gfp_allowed_mask;
> >>
> >> @@ -2543,6 +2544,13 @@ __alloc_pages_nodemask(gfp_t gfp_mask, unsigned int order,
> >>   return NULL;
> >>
> >> /*
> >> + * Will only have any effect when __GFP_KMEMCG is set.
> >> + * This is verified in the (always inline) callee
> >> + */
> >> + if (!memcg_kmem_new_page(gfp_mask, &handle, order))
> > >
> > > memcg_kmem_new_page takes a void * parameter already but here you are
> > > passing in a void **. This probably happens to work because you do this
> > >
> > > struct mem_cgroup **handle = (struct mem_cgroup **)_handle;
> > >
> > > but that appears to defeat the purpose of having an opaque type as a
> > > "handle". You have to treat it different then passing it into the commit
> > > function because it expects a void *. The motivation for an opaque type
> > > is completely unclear to me and how it is managed with a mix of void *
> > > and void ** is very confusing.
>
> okay.
>
> The opaque exists because I am doing speculative charging.

I do not get why speculative charing would mandate an opaque type or
"handle". It looks like like a fairly standard prepare/commit pattern to me.

> I believe it
> to be a better and less complicated approach then letting a page appear
> and then charging it. Besides being consistent with the rest of memcg,
> it won't create unnecessary disturbance in the page allocator
> when the allocation is to fail.

>

I still don't get why you did not just return a mem_cgroup instead of a
handle.

> Now, tasks can move between memcgs, so we can't rely on grabbing it from
> current in commit_page, so we pass it around as a handle.

You could just as easily passed around the mem_cgroup and it would have
been less obscure. Maybe this makes sense from a memcg context and matches
some coding pattern there that I'm not aware of.

> Also, even if
> the task could not move, we already got it once from the task, and that
> is not for free. Better save it.
>
> Aside from the handle needed, the cost is more or less the same compared
> to doing it in one pass. All we do by using speculative charging is to
> split the cost in two, and doing it from two places.
> We'd have to charge + update page_cgroup anyway.
>
> As for the type, do you think using struct mem_cgroup would be less
> confusing?
>

Yes and returning the mem_cgroup or NULL instead of bool.

> > On a similar note I spotted #define memcg_kmem_on 1 . That is also
> > different just for the sake of it. The convension is to do something
> > like this
> >
> > /* This helps us to avoid #ifdef CONFIG_NUMA */
> > #ifdef CONFIG_NUMA
> > #define NUMA_BUILD 1
> > #else
> > #define NUMA_BUILD 0
> > #endif
>
> For simple defines, yes. But a later patch will turn this into a static
> branch test. memcg_kmem_on will be always 0 when compile-disabled, but
> when enable will expand to static_branch(&...).
>

I see.

>
> > memcg_kmem_on was difficult to guess based on its name. I thought initially
> > that it would only be active if a memcg existed or at least something like

> > mem_cgroup_disabled() but it's actually enabled if CONFIG_MEMCG_KMEM is set.
>
> For now. And I thought that adding the static branch in this patch would
> only confuse matters.

Ah, I see now. I had stopped reading the series once I reached this
patch. I don't think it would have mattered much to collapse the two
patches together but ok.

The static key handling does look a little suspicious. You appear to do
reference counting in memcg_update_kmem_limit for every mem_cgroup_write()
but decrement it on memcg exit. This does not appear as if it would be
symmetric if the memcg files were written to multiple times (maybe that's
not allowed?). Either way, the comment says it can never be disabled but
as you have static_key_slow_dec calls it would appear that you *do*
support them being disabled. Confusing.

> The placeholder is there, but it is later patched
> to the final thing.
> With that explained, if you want me to change it to something else, I
> can do it. Should I ?
>

Not in this patch anyway. I would have preferred a pattern like this but
that's about it.

```
#ifdef CONFIG_MEMCG_KMEM
extern struct static_key memcg_kmem_enabled_key;
static inline int memcg_kmem_enabled(void)
{
	return static_key_false(&memcg_kmem_enabled_key);
}
#else

static inline bool memcg_kmem_enabled(void)
{
	return false;
}
#endif
```

Two reasons. One, it does not use the terms "on" and "enabled"
interchangeably. The other reason is down to taste as I'm copying the
pattern I used myself for sk_memalloc_socks(). Of course I am biased.

Also, why is the key exported?

> > I also find it *very* strange to have a function named as if it is an
> > allocation-style function when it in fact it's looking up a mem_cgroup

> > and charging it (and uncharging it in the error path if necessary). If
> > it was called memcg_kmem_newpage_charge I might have found it a little
> > better.
>
> I don't feel strongly about names in general. I can change it.
> Will update to memcg_kmem_newpage_charge() and memcg_kmem_page_uncharge().
>

I would prefer that anyway. Names have meaning and people make assumptions on
the implementation depending on the name. We should try to be as consistent
as possible or maintenance becomes harder. I know there are areas where
we are not consistent at all but we should not compound the problem.

> > This whole operation also looks very expensive (cgroup lookups, RCU locks
> > taken etc) but I guess you're willing to take that cost in the same of
> > isolating containers from each other. However, I strongly suggest that
> > this overhead is measured in advance. It should not stop the series being
> > merged as such but it should be understood because if the cost is high
> > then this feature will be avoided like the plague. I am skeptical that
> > distributions would enable this by default, at least not without support
> > for cgroup_disable=kmem
>
> Enabling this feature will bring you nothing, therefore, no (or little)
> overhead. Nothing of this will be patched in until the first memcg gets
> kmem limited. The mere fact of moving tasks to memcgs won't trigger any
> of this.
>

ok.

> I haven't measured this series in particular, but I did measure the slab
> series (which builds ontop of this). I found the per-allocation cost to
> be in the order of 2-3 % for tasks living in limited memcgs, and
> hard to observe when living in the root memcg (compared of course to the
> case of a task running on root memcg without those patches)
>

Depending on the workload that 2-3% could be a lot but at least you're
aware of it.

> I also believe the folks from google also measured this. They may be
> able to spit out numbers grabbed from a system bigger than mine =p
>
> > As this thing is called from within the allocator, it's not clear why
> > __memcg_kmem_new_page is exported. I can't imagine why a module would call
> > it directly although maybe you cover that somewhere else in the series.
>
> Okay, more people commented on this, so let me clarify: They shouldn't

> be. They were initially exported when this was about the slab only,
> because they could be called from inlined functions from the allocators.
> Now that the charge/uncharge was moved to the page allocator - which
> already allowed me the big benefit of separating this in two pieces,
> none of this needs to be exported.
>
> Sorry for not noticing this myself, but thanks for the eyes =)
>

You're welcome. I expect to see all the exports disappear so. If there
are any exports left I think it would be important to document why they
have to be exported. This is particularly true because they are
EXPORT_SYMBOL not EXPORT_SYMBOL_GPL. I think it would be good to know in
advance why a module (particularly an out-of-tree one) would be
interested.

> > From the point of view of a hook, that is acceptable but just barely. I have
> > slammed other hooks because it was possible for a subsystem to override them
> > meaning the runtime cost could be anything. I did not spot a similar issue
> > here but if I missed it, it's still unacceptable. At least here the cost
> > is sortof predictable and only affects memcg because of the __GFP_KMEMCG
> > check in memcg_kmem_new_page.
>
> Yes, that is the idea. And I don't think anyone should override those,
> so I don't see them as hooks in this sense.
>

Indeed not, callbacks are the real issue.

> >> + return NULL;
> >> +
> >> + /*
> >>   * Check the zones suitable for the gfp_mask contain at least one
> >>   * valid zone. It's possible to have an empty zonelist as a result
> >>   * of GFP_THISNODE and a memoryless node
> >> @@ -2583,6 +2591,8 @@ out:
> >>   if (unlikely(!put_mems_allowed(cpuset_mems_cookie) && !page))
> >>   goto retry_cpuset;
> >>
> >> + memcg_kmem_commit_page(page, handle, order);
> >> +
> >
> > As a side note, I'm not keen on how you shortcut these functions. They
> > are all function calls because memcg_kmem_commit_page() will always call
> > __memcg_kmem_commit_page() to check the handle once it's compiled in.
> > The handle==NULL check should have happened in the inline function to save
> > a few cycles.
> >

>
> It is already happening on my updated series after a comment from Kame
> pointed this out.
>

ok.

> > <SNIP>
> >
> > memcg_kmem_new_page makes the following check
> >
> > +	if (!(gfp & __GFP_KMEMCG) || (gfp & __GFP_NOFAIL))
> > +		return true;
> >
> > so if the allocation had __GFP_NOFAIL, it does not get charged but can
> > still be freed. I didn't check if this is really the case but it looks
> > very suspicious.
>
> No, it can't be freed (uncharged), because in that case, we won't fill
> in the memcg information in page cgroup.
>

Ah, I see.

> > Again, this is a fairly heavy operation.
>
>
> Mel, once I address all the issues you pointed out here, do you think
> this would be in an acceptable state for merging? Do you still have any
> fundamental opposition to this?
>

I do not have a fundamental opposition to it, particularly as it only
has an impact when it's enabled. This is not an ack either though as I
see the series in general still has a lot of feedback outstanding
including this patch.

--
Mel Gorman
SUSE Labs

Subject: Re: [PATCH v2 04/11] kmem accounting basic infrastructure
Posted by Michal Hocko on Wed, 15 Aug 2012 13:26:21 GMT
View Forum Message <> Reply to Message

On Wed 15-08-12 17:04:31, Glauber Costa wrote:
> On 08/15/2012 05:02 PM, Michal Hocko wrote:

> > On Wed 15-08-12 16:53:40, Glauber Costa wrote:
> > [...]
> >>>>> This doesn't check for the hierachy so kmem_accounted might not be in
> >>>>> sync with it's parents. mem_cgroup_create (below) needs to copy
> >>>>> kmem_accounted down from the parent and the above needs to check if this
> >>>>> is a similar dance like mem_cgroup_oom_control_write.
> >>>>>
> >>>>
> >>>> I don't see why we have to.
> >>>>
> >>>> I believe in a A/B/C hierarchy, C should be perfectly able to set a
> >>>> different limit than its parents. Note that this is not a boolean.
> >>>
> >>> Ohh, I wasn't clear enough. I am not against setting the _limit_ I just
> >>> meant that the kmem_accounted should be consistent within the hierarchy.
> >>>
> >>
> >> If a parent of yours is accounted, you get accounted as well. This is
> >> not the state in this patch, but gets added later. Isn't this enough ?
> >
> > But if the parent is not accounted, you can set the children to be
> > accounted, right? Or maybe this is changed later in the series? I didn't
> > get to the end yet.
> >
>
> Yes, you can. Do you see any problem with that?

Well, if a child contributes with the kmem charges upwards the hierachy
then a parent can have kmem.usage > 0 with disabled accounting.
I am not saying this is a no-go but it definitely is confusing and I do
not see any good reason for it. I've considered it as an overlook rather
than a deliberate design decision.
--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v2 04/11] kmem accounting basic infrastructure
Posted by James Bottomley on Wed, 15 Aug 2012 13:29:57 GMT
View Forum Message <> Reply to Message

On Wed, 2012-08-15 at 14:55 +0200, Michal Hocko wrote:
> On Wed 15-08-12 12:12:23, James Bottomley wrote:
> > On Wed, 2012-08-15 at 13:33 +0400, Glauber Costa wrote:
> > > > > This can
> > > > > be quite confusing. I am still not sure whether we should mix the two
> > > > > things together. If somebody wants to limit the kernel memory he has to
> > > > > touch the other limit anyway. Do you have a strong reason to mix the

> > > > user and kernel counters?
> > >
> > > This is funny, because the first opposition I found to this work was
> > > "Why would anyone want to limit it separately?" =p
> > >
> > > It seems that a quite common use case is to have a container with a
> > > unified view of "memory" that it can use the way he likes, be it with
> > > kernel memory, or user memory. I believe those people would be happy to
> > > just silently account kernel memory to user memory, or at the most have
> > > a switch to enable it.
> > >
> > > What gets clear from this back and forth, is that there are people
> > > interested in both use cases.
> >
> > Haven't we already had this discussion during the Prague get together?
> > We discussed the use cases and finally agreed to separate accounting for
> > k and then k+u mem because that satisfies both the Google and Parallels
> > cases. No-one was overjoyed by k and k+u but no-one had a better
> > suggestion ... is there a better way of doing this that everyone can
> > agree to?
> > We do need to get this nailed down because it's the foundation of the
> > patch series.
>
> There is a slot in MM/memcg minisum at KS so we have a slot to discuss
> this.

Sure, to get things moving, can you pre-prime us with what you're
thinking in this area so we can be prepared (and if it doesn't work,
tell you beforehand)?

Thanks,

James

Subject: Re: [PATCH v2 04/11] kmem accounting basic infrastructure
Posted by Glauber Costa on Wed, 15 Aug 2012 13:31:24 GMT
View Forum Message <> Reply to Message

On 08/15/2012 05:26 PM, Michal Hocko wrote:
> On Wed 15-08-12 17:04:31, Glauber Costa wrote:
>> On 08/15/2012 05:02 PM, Michal Hocko wrote:
>>> On Wed 15-08-12 16:53:40, Glauber Costa wrote:
>>> [...]
>>>>>> This doesn't check for the hierachy so kmem_accounted might not be in
>>>>>> sync with it's parents. mem_cgroup_create (below) needs to copy
>>>>>> kmem_accounted down from the parent and the above needs to check if this
>>>>>> is a similar dance like mem_cgroup_oom_control_write.

>>>>>>>
>>>>>>
>>>>>> I don't see why we have to.
>>>>>>
>>>>>> I believe in a A/B/C hierarchy, C should be perfectly able to set a
>>>>>> different limit than its parents. Note that this is not a boolean.
>>>>>
>>>>> Ohh, I wasn't clear enough. I am not against setting the _limit_ I just
>>>>> meant that the kmem_accounted should be consistent within the hierarchy.
>>>>>
>>>>
>>>> If a parent of yours is accounted, you get accounted as well. This is
>>>> not the state in this patch, but gets added later. Isn't this enough ?
>>>
>>> But if the parent is not accounted, you can set the children to be
>>> accounted, right? Or maybe this is changed later in the series? I didn't
>>> get to the end yet.
>>>
>>
>> Yes, you can. Do you see any problem with that?
>
> Well, if a child contributes with the kmem charges upwards the hierachy
> then a parent can have kmem.usage > 0 with disabled accounting.
> I am not saying this is a no-go but it definitely is confusing and I do
> not see any good reason for it. I've considered it as an overlook rather
> than a deliberate design decision.
>

No, it is not an overlook.
It is theoretically possible to skip accounting on non-limited parents,
but how expensive is that? This is, indeed, confusing.

Of course I can be biased, but the way I see it, once you have
hierarchy, you account everything your child accounts.

I really don't see what is the concern here.

Subject: Re: [PATCH v2 07/11] mm: Allocate kernel pages to the right memcg
Posted by Glauber Costa on Wed, 15 Aug 2012 13:39:58 GMT
View Forum Message <> Reply to Message

>>
>> As for the type, do you think using struct mem_cgroup would be less
>> confusing?
>>
>
> Yes and returning the mem_cgroup or NULL instead of bool.

Ok. struct mem_cgroup it is.

>
>> The placeholder is there, but it is later patched
>> to the final thing.
>> With that explained, if you want me to change it to something else, I
>> can do it. Should I ?
>>
>
> Not in this patch anyway. I would have preferred a pattern like this but
> that's about it.
>
> #ifdef CONFIG_MEMCG_KMEM
> extern struct static_key memcg_kmem_enabled_key;
> static inline int memcg_kmem_enabled(void)
> {
> return static_key_false(&memcg_kmem_enabled_key);
> }
> #else
>
> static inline bool memcg_kmem_enabled(void)
> {
> return false;
> }
> #endif
>

humm, I'll have to think about this name.
"memcg_kmem_enabled" means it is enabled in this cgroup. It is actually
used inside memcontrol.c to denote precisely that.

Now the static branch, of course, means it is globally enabled. Or as I
called here, "on".

> Two reasons. One, it does not use the terms "on" and "enabled"
> interchangeably. The other reason is down to taste as I'm copying the
> pattern I used myself for sk_memalloc_socks(). Of course I am biased.
>
> Also, why is the key exported?
>

Same reason. The slab will now have inline functions that will test
against that. The alloc functions themselves, are inside the page
allocator, and the exports can go away.

But the static branch will still be tested inside inlined functions in

the slab.

That said, for the sake of simplicity, I can make it go away here, and
add that to the right place later.

>>> I also find it *very* strange to have a function named as if it is an
>>> allocation-style function when it in fact it's looking up a mem_cgroup
>>> and charging it (and uncharging it in the error path if necessary). If
>>> it was called memcg_kmem_newpage_charge I might have found it a little
>>> better.
>>
>> I don't feel strongly about names in general. I can change it.
>> Will update to memcg_kmem_newpage_charge() and memcg_kmem_page_uncharge().
>>
>
> I would prefer that anyway. Names have meaning and people make assumptions on
> the implementation depending on the name. We should try to be as consistent
> as possible or maintenance becomes harder. I know there are areas where
> we are not consistent at all but we should not compound the problem.

memcg_kmem_page_charge() is even better I believe, and that is what I
changed this to in my tree.

>>> As this thing is called from within the allocator, it's not clear why
>>> __memcg_kmem_new_page is exported. I can't imagine why a module would call
>>> it directly although maybe you cover that somewhere else in the series.
>>
>> Okay, more people commented on this, so let me clarify: They shouldn't
>> be. They were initially exported when this was about the slab only,
>> because they could be called from inlined functions from the allocators.
>> Now that the charge/uncharge was moved to the page allocator - which
>> already allowed me the big benefit of separating this in two pieces,
>> none of this needs to be exported.
>>
>> Sorry for not noticing this myself, but thanks for the eyes =)
>>
>
> You're welcome. I expect to see all the exports disappear so. If there
> are any exports left I think it would be important to document why they
> have to be exported. This is particularly true because they are
> EXPORT_SYMBOL not EXPORT_SYMBOL_GPL. I think it would be good to know in
> advance why a module (particularly an out-of-tree one) would be
> interested.
>

I will remove them all for now.

Subject: Re: [PATCH v2 07/11] mm: Allocate kernel pages to the right memcg
Posted by Glauber Costa on Wed, 15 Aug 2012 13:51:40 GMT
View Forum Message <> Reply to Message

On 08/15/2012 05:22 PM, Mel Gorman wrote:
>> I believe it
>> > to be a better and less complicated approach then letting a page appear
>> > and then charging it. Besides being consistent with the rest of memcg,
>> > it won't create unnecessary disturbance in the page allocator
>> > when the allocation is to fail.
>> >
> I still don't get why you did not just return a mem_cgroup instead of a
> handle.
>

Forgot this one, sorry:

The reason is to keep the semantics simple.

What should we return if the code is not compiled in? If we return NULL
for failure, the test becomes

```
memcg = memcg_kmem_charge_page(gfp, order);
if (!memcg)
  exit;
```

If we're not compiled in, we'd either return positive garbage or we need
to wrap it inside an ifdef

I personally believe to be a lot more clear to standardize on true
to mean "allocation can proceed".

the compiled out case becomes:

```
if (!true)
   exit;
```

which is easily compiled away altogether. Now of course, using struct
mem_cgroup makes sense, and I have already changed that here.

Subject: Re: [PATCH v2 06/11] memcg: kmem controller infrastructure
Posted by Glauber Costa on Wed, 15 Aug 2012 14:01:51 GMT
View Forum Message <> Reply to Message

On 08/15/2012 05:09 PM, Michal Hocko wrote:
> On Wed 15-08-12 13:42:24, Glauber Costa wrote:
> [...]

>>>> +
>>>> + ret = 0;
>>>> +
>>>> + if (!memcg)
>>>> + return ret;
>>>> +
>>>> + _memcg = memcg;
>>>> + ret = __mem_cgroup_try_charge(NULL, gfp, delta / PAGE_SIZE,
>>>> +   &_memcg, may_oom);
>>>
>>> This is really dangerous because atomic allocation which seem to be
>>> possible could result in deadlocks because of the reclaim.
>>
>> Can you elaborate on how this would happen?
>
> Say you have an atomic allocation and we hit the limit so we get either
> to reclaim which can sleep or to oom which can sleep as well (depending
> on the oom_control).
>

I see now, you seem to be right.

How about we change the following code in mem_cgroup_do_charge:

```
	if (gfp_mask & __GFP_NORETRY)
		return CHARGE_NOMEM;
```

to:

```
	if ((gfp_mask & __GFP_NORETRY) || (gfp_mask & __GFP_ATOMIC))
		return CHARGE_NOMEM;
```

?

Would this take care of the issue ?

Subject: Re: [PATCH v2 04/11] kmem accounting basic infrastructure
Posted by Michal Hocko on Wed, 15 Aug 2012 14:10:41 GMT
View Forum Message <> Reply to Message

On Wed 15-08-12 17:31:24, Glauber Costa wrote:
> On 08/15/2012 05:26 PM, Michal Hocko wrote:
> > On Wed 15-08-12 17:04:31, Glauber Costa wrote:
> >> On 08/15/2012 05:02 PM, Michal Hocko wrote:
> >>> On Wed 15-08-12 16:53:40, Glauber Costa wrote:
> >>> [...]
> >>>>>>> This doesn't check for the hierachy so kmem_accounted might not be in

> >>>>>>> sync with it's parents. mem_cgroup_create (below) needs to copy
> >>>>>>> kmem_accounted down from the parent and the above needs to check if this
> >>>>>>> is a similar dance like mem_cgroup_oom_control_write.
> >>>>>>>
> >>>>>>
> >>>>>> I don't see why we have to.
> >>>>>>
> >>>>>> I believe in a A/B/C hierarchy, C should be perfectly able to set a
> >>>>>> different limit than its parents. Note that this is not a boolean.
> >>>>>
> >>>>> Ohh, I wasn't clear enough. I am not against setting the _limit_ I just
> >>>>> meant that the kmem_accounted should be consistent within the hierarchy.
> >>>>>
> >>>>
> >>>> If a parent of yours is accounted, you get accounted as well. This is
> >>>> not the state in this patch, but gets added later. Isn't this enough ?
> >>>
> >>> But if the parent is not accounted, you can set the children to be
> >>> accounted, right? Or maybe this is changed later in the series? I didn't
> >>> get to the end yet.
> >>>
> >>
> >> Yes, you can. Do you see any problem with that?
> >
> > Well, if a child contributes with the kmem charges upwards the hierachy
> > then a parent can have kmem.usage > 0 with disabled accounting.
> > I am not saying this is a no-go but it definitely is confusing and I do
> > not see any good reason for it. I've considered it as an overlook rather
> > than a deliberate design decision.
> >
>
> No, it is not an overlook.
> It is theoretically possible to skip accounting on non-limited parents,
> but how expensive is that? This is, indeed, confusing.
>
> Of course I can be biased, but the way I see it, once you have
> hierarchy, you account everything your child accounts.
>
> I really don't see what is the concern here.

OK, I missed an important point that kmem_accounted is not exported to
the userspace (I thought it would be done later in the series) which
is not the case so actually nobody get's confused by the inconsistency
because it is about RESOURCE_MAX which they see in both cases.
Sorry about the confusion!
--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v2 04/11] kmem accounting basic infrastructure
Posted by Glauber Costa on Wed, 15 Aug 2012 14:11:41 GMT
View Forum Message <> Reply to Message

> OK, I missed an important point that kmem_accounted is not exported to
> the userspace (I thought it would be done later in the series) which
> is not the case so actually nobody get's confused by the inconsistency
> because it is about RESOURCE_MAX which they see in both cases.
> Sorry about the confusion!
>
I'll forgive you this time...

Subject: Re: [PATCH v2 06/11] memcg: kmem controller infrastructure
Posted by Michal Hocko on Wed, 15 Aug 2012 14:23:38 GMT
View Forum Message <> Reply to Message

On Wed 15-08-12 18:01:51, Glauber Costa wrote:
> On 08/15/2012 05:09 PM, Michal Hocko wrote:
> > On Wed 15-08-12 13:42:24, Glauber Costa wrote:
> > [...]
> >>>> +
> >>>> + ret = 0;
> >>>> +
> >>>> + if (!memcg)
> >>>> + return ret;
> >>>> +
> >>>> + _memcg = memcg;
> >>>> + ret = __mem_cgroup_try_charge(NULL, gfp, delta / PAGE_SIZE,
> >>>> +   &_memcg, may_oom);
> >>>
> >>> This is really dangerous because atomic allocation which seem to be
> >>> possible could result in deadlocks because of the reclaim.
> >>
> >> Can you elaborate on how this would happen?
> >
> > Say you have an atomic allocation and we hit the limit so we get either
> > to reclaim which can sleep or to oom which can sleep as well (depending
> > on the oom_control).
> >
>
> I see now, you seem to be right.

No I am not because it seems that I am really blind these days...
We were doing this in mem_cgroup_do_charge for ages:
 if (!(gfp_mask & __GFP_WAIT))
  return CHARGE_WOULDBLOCK;

/me goes to hide and get with further feedback with a clean head.

Sorry about that.
--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v2 06/11] memcg: kmem controller infrastructure
Posted by Glauber Costa on Wed, 15 Aug 2012 14:27:45 GMT
View Forum Message <> Reply to Message

>>
>> I see now, you seem to be right.
>
> No I am not because it seems that I am really blind these days...
> We were doing this in mem_cgroup_do_charge for ages:
> if (!(gfp_mask & __GFP_WAIT))
> return CHARGE_WOULDBLOCK;
>
> /me goes to hide and get with further feedback with a clean head.
>
> Sorry about that.
>
I am as well, since I went to look at mem_cgroup_do_charge() and missed
that.

Do you have any other concerns specific to this patch ?

Subject: Re: [PATCH v2 04/11] kmem accounting basic infrastructure
Posted by Christoph Lameter on Wed, 15 Aug 2012 14:47:57 GMT
View Forum Message <> Reply to Message

On Wed, 15 Aug 2012, Michal Hocko wrote:

> > That is not what the kernel does, in general. We assume that if he wants
> > that memory and we can serve it, we should. Also, not all kernel memory
> > is unreclaimable. We can shrink the slabs, for instance. Ying Han
> > claims she has patches for that already...
>
> Are those patches somewhere around?

You can already shrink the reclaimable slabs (dentries / inodes) via
calls to the subsystem specific shrinkers. Did Ying Han do anything to
go beyond that?

Subject: Re: [PATCH v2 04/11] kmem accounting basic infrastructure
Posted by Glauber Costa on Wed, 15 Aug 2012 15:11:49 GMT
View Forum Message <> Reply to Message

On 08/15/2012 06:47 PM, Christoph Lameter wrote:
> On Wed, 15 Aug 2012, Michal Hocko wrote:
>
>>> That is not what the kernel does, in general. We assume that if he wants
>>> that memory and we can serve it, we should. Also, not all kernel memory
>>> is unreclaimable. We can shrink the slabs, for instance. Ying Han
>>> claims she has patches for that already...
>>
>> Are those patches somewhere around?
>
> You can already shrink the reclaimable slabs (dentries / inodes) via
> calls to the subsystem specific shrinkers. Did Ying Han do anything to
> go beyond that?
>
That is not enough for us.
We would like to make sure that the objects being discarded belong to
the memcg which is under pressure. We don't need to be perfect here, and
an occasional slip is totally fine. But if in general, shrinking from
memcg A will mostly wipe out objects from memcg B, we harmed the system
in return for nothing good.

Subject: Re: [PATCH v2 04/11] kmem accounting basic infrastructure
Posted by Greg Thelen on Wed, 15 Aug 2012 15:19:58 GMT
View Forum Message <> Reply to Message

On Wed, Aug 15 2012, Christoph Lameter wrote:

> On Wed, 15 Aug 2012, Michal Hocko wrote:
>
>> > That is not what the kernel does, in general. We assume that if he wants
>> > that memory and we can serve it, we should. Also, not all kernel memory
>> > is unreclaimable. We can shrink the slabs, for instance. Ying Han
>> > claims she has patches for that already...
>>
>> Are those patches somewhere around?
>
> You can already shrink the reclaimable slabs (dentries / inodes) via
> calls to the subsystem specific shrinkers. Did Ying Han do anything to
> go beyond that?

cc: Ying

The Google shrinker patches enhance prune_dcache_sb() to limit dentry

pressure to a specific memcg.

Subject: Re: [PATCH v2 04/11] kmem accounting basic infrastructure
Posted by Christoph Lameter on Wed, 15 Aug 2012 15:34:39 GMT
View Forum Message <> Reply to Message

On Wed, 15 Aug 2012, Glauber Costa wrote:

> On 08/15/2012 06:47 PM, Christoph Lameter wrote:
> > On Wed, 15 Aug 2012, Michal Hocko wrote:
> >
> >>> That is not what the kernel does, in general. We assume that if he wants
> >>> that memory and we can serve it, we should. Also, not all kernel memory
> >>> is unreclaimable. We can shrink the slabs, for instance. Ying Han
> >>> claims she has patches for that already...
> >>
> >> Are those patches somewhere around?
> >
> > You can already shrink the reclaimable slabs (dentries / inodes) via
> > calls to the subsystem specific shrinkers. Did Ying Han do anything to
> > go beyond that?
> >
> That is not enough for us.
> We would like to make sure that the objects being discarded belong to
> the memcg which is under pressure. We don't need to be perfect here, and
> an occasional slip is totally fine. But if in general, shrinking from
> memcg A will mostly wipe out objects from memcg B, we harmed the system
> in return for nothing good.

How can you figure out which objects belong to which memcg? The ownerships
of dentries and inodes is a dubious concept already.

Subject: Re: [PATCH v2 04/11] kmem accounting basic infrastructure
Posted by Glauber Costa on Wed, 15 Aug 2012 15:35:13 GMT
View Forum Message <> Reply to Message

On 08/15/2012 07:34 PM, Christoph Lameter wrote:
> On Wed, 15 Aug 2012, Glauber Costa wrote:
>
>> On 08/15/2012 06:47 PM, Christoph Lameter wrote:
>>> On Wed, 15 Aug 2012, Michal Hocko wrote:
>>>
>>>>> That is not what the kernel does, in general. We assume that if he wants
>>>>> that memory and we can serve it, we should. Also, not all kernel memory
>>>>> is unreclaimable. We can shrink the slabs, for instance. Ying Han

>>>>> claims she has patches for that already...
>>>>
>>>> Are those patches somewhere around?
>>>
>>> You can already shrink the reclaimable slabs (dentries / inodes) via
>>> calls to the subsystem specific shrinkers. Did Ying Han do anything to
>>> go beyond that?
>>>
>> That is not enough for us.
>> We would like to make sure that the objects being discarded belong to
>> the memcg which is under pressure. We don't need to be perfect here, and
>> an occasional slip is totally fine. But if in general, shrinking from
>> memcg A will mostly wipe out objects from memcg B, we harmed the system
>> in return for nothing good.
>
> How can you figure out which objects belong to which memcg? The ownerships
> of dentries and inodes is a dubious concept already.
>

Remember we copy over the metadata and create copies of the caches
per-memcg. Therefore, a dentry belongs to a memcg if it was allocated
from the slab pertaining to that memcg.

It is not 100 % accurate, but it is good enough.

Subject: Re: [PATCH v2 04/11] kmem accounting basic infrastructure
Posted by Christoph Lameter on Wed, 15 Aug 2012 15:36:28 GMT
View Forum Message <> Reply to Message

On Wed, 15 Aug 2012, Greg Thelen wrote:

> > You can already shrink the reclaimable slabs (dentries / inodes) via
> > calls to the subsystem specific shrinkers. Did Ying Han do anything to
> > go beyond that?
>
> cc: Ying
>
> The Google shrinker patches enhance prune_dcache_sb() to limit dentry
> pressure to a specific memcg.

Ok then its restricted to the reclaimable slab caches already. The main
issue to sort out then is who is the "owner" of an inode/dentry (if
something like that exists). If you separate the objects into different
pages then the objects may be cleanly separated at the price of more
memory use.

Subject: Re: [PATCH v2 06/11] memcg: kmem controller infrastructure
Posted by Greg Thelen on Wed, 15 Aug 2012 16:38:57 GMT
View Forum Message <> Reply to Message

On Wed, Aug 15 2012, Glauber Costa wrote:

> On 08/14/2012 10:58 PM, Greg Thelen wrote:
>> On Mon, Aug 13 2012, Glauber Costa wrote:
>>
>>>>>> + WARN_ON(mem_cgroup_is_root(memcg));
>>>>>> + size = (1 << order) << PAGE_SHIFT;
>>>>>> + memcg_uncharge_kmem(memcg, size);
>>>>>> + mem_cgroup_put(memcg);
>>>> Why do we need ref-counting here ? kmem res_counter cannot work as
>>>> reference ?
>>> This is of course the pair of the mem_cgroup_get() you commented on
>>> earlier. If we need one, we need the other. If we don't need one, we
>>> don't need the other =)
>>>
>>> The guarantee we're trying to give here is that the memcg structure will
>>> stay around while there are dangling charges to kmem, that we decided
>>> not to move (remember: moving it for the stack is simple, for the slab
>>> is very complicated and ill-defined, and I believe it is better to treat
>>> all kmem equally here)
>>
>> By keeping memcg structures hanging around until the last referring kmem
>> page is uncharged do such zombie memcg each consume a css_id and thus
>> put pressure on the 64k css_id space? I imagine in pathological cases
>> this would prevent creation of new cgroups until these zombies are
>> dereferenced.
>
> Yes, but although this patch makes it more likely, it doesn't introduce
> that. If the tasks, for instance, grab a reference to the cgroup dentry
> in the filesystem (like their CWD, etc), they will also keep the cgroup
> around.

Fair point. But this doesn't seems like a feature. It's probably not needed initially, but what do you think about creating a memcg_kernel_context structure which is allocated when memcg is allocated? Kernel pages charged to a memcg would have page_cgroup->mem_cgroup=memcg_kernel_context rather than memcg. This would allow the mem_cgroup and its css_id to be deleted when the cgroup is unlinked from cgroupfs while allowing for the active kernel pages to continue pointing to a valid memcg_kernel_context. This would be a reference counted structure much like you are doing with memcg. When a memcg is deleted the memcg_kernel_context would be linked into its surviving parent memcg. This would avoid needing to visit each kernel page.

>> Is there any way to see how much kmem such zombie memcg are consuming?
>> I think we could find these with
>> for_each_mem_cgroup_tree(root_mem_cgroup).
>
> Yes, just need an interface for that. But I think it is something that
> can be addressed orthogonaly to this work, in a separate patch, not as
> some fundamental limitation.

Agreed.

>> Basically, I'm wanting to know where kernel memory has been
>> allocated. For live memcg, an admin can cat
>> memory.kmem.usage_in_bytes. But for zombie memcg, I'm not sure how
>> to get this info. It looks like the root_mem_cgroup
>> memory.kmem.usage_in_bytes is not hierarchically charged.
>>
>
> Not sure what you mean by not being hierarchically charged. It should
> be, when use_hierarchy = 1. As a matter of fact, I just tested it, and I
> do see kmem being charged all the way to the root cgroup when hierarchy
> is used. (we just can't limit it there)

You're correct, my mistake.

I think the procedure to determine out the amount of zombie kmem is:
  root_mem_cgroup.kmem_usage_in_bytes -
    sum(all top level memcg memory.kmem_usage_in_bytes)

Subject: Re: [PATCH v2 06/11] memcg: kmem controller infrastructure
Posted by Glauber Costa on Wed, 15 Aug 2012 17:00:31 GMT
View Forum Message <> Reply to Message

On 08/15/2012 08:38 PM, Greg Thelen wrote:
> On Wed, Aug 15 2012, Glauber Costa wrote:
>
>> On 08/14/2012 10:58 PM, Greg Thelen wrote:
>>> On Mon, Aug 13 2012, Glauber Costa wrote:
>>>
>>>>>> + WARN_ON(mem_cgroup_is_root(memcg));
>>>>>> + size = (1 << order) << PAGE_SHIFT;
>>>>>> + memcg_uncharge_kmem(memcg, size);
>>>>>> + mem_cgroup_put(memcg);
>>>>> Why do we need ref-counting here ? kmem res_counter cannot work as
>>>>> reference ?
>>>> This is of course the pair of the mem_cgroup_get() you commented on
>>>> earlier. If we need one, we need the other. If we don't need one, we
>>>> don't need the other =)

>>>>
>>>> The guarantee we're trying to give here is that the memcg structure will
>>>> stay around while there are dangling charges to kmem, that we decided
>>>> not to move (remember: moving it for the stack is simple, for the slab
>>>> is very complicated and ill-defined, and I believe it is better to treat
>>>> all kmem equally here)
>>>
>>> By keeping memcg structures hanging around until the last referring kmem
>>> page is uncharged do such zombie memcg each consume a css_id and thus
>>> put pressure on the 64k css_id space? I imagine in pathological cases
>>> this would prevent creation of new cgroups until these zombies are
>>> dereferenced.
>>
>> Yes, but although this patch makes it more likely, it doesn't introduce
>> that. If the tasks, for instance, grab a reference to the cgroup dentry
>> in the filesystem (like their CWD, etc), they will also keep the cgroup
>> around.
>
> Fair point. But this doesn't seems like a feature. It's probably not
> needed initially, but what do you think about creating a
> memcg_kernel_context structure which is allocated when memcg is
> allocated? Kernel pages charged to a memcg would have
> page_cgroup->mem_cgroup=memcg_kernel_context rather than memcg. This
> would allow the mem_cgroup and its css_id to be deleted when the cgroup
> is unlinked from cgroupfs while allowing for the active kernel pages to
> continue pointing to a valid memcg_kernel_context. This would be a
> reference counted structure much like you are doing with memcg. When a
> memcg is deleted the memcg_kernel_context would be linked into its
> surviving parent memcg. This would avoid needing to visit each kernel
> page.

You need more, you need at the res_counters to stay around as well. And
probably other fields.

So my fear here is that as you add fields to that structure, you can
defeat a bit the goal of reducing memory consumption. Still leaves the
css space, yes. But by doing this we can introduce some subtle bugs by
having a field in the wrong structure.

Did you observe that to be a big problem in your systems?

Subject: Re: [PATCH v2 06/11] memcg: kmem controller infrastructure
Posted by Greg Thelen on Wed, 15 Aug 2012 17:12:21 GMT
View Forum Message <> Reply to Message

On Wed, Aug 15 2012, Glauber Costa wrote:

> On 08/15/2012 08:38 PM, Greg Thelen wrote:
>> On Wed, Aug 15 2012, Glauber Costa wrote:
>>
>>> On 08/14/2012 10:58 PM, Greg Thelen wrote:
>>>> On Mon, Aug 13 2012, Glauber Costa wrote:
>>>>
>>>>>>>>>> + WARN_ON(mem_cgroup_is_root(memcg));
>>>>>>>>>> + size = (1 << order) << PAGE_SHIFT;
>>>>>>>>>> + memcg_uncharge_kmem(memcg, size);
>>>>>>>>>> + mem_cgroup_put(memcg);
>>>>>>> Why do we need ref-counting here ? kmem res_counter cannot work as
>>>>>>> reference ?
>>>>>> This is of course the pair of the mem_cgroup_get() you commented on
>>>>>> earlier. If we need one, we need the other. If we don't need one, we
>>>>>> don't need the other =)
>>>>>>
>>>>>> The guarantee we're trying to give here is that the memcg structure will
>>>>>> stay around while there are dangling charges to kmem, that we decided
>>>>>> not to move (remember: moving it for the stack is simple, for the slab
>>>>>> is very complicated and ill-defined, and I believe it is better to treat
>>>>>> all kmem equally here)
>>>>
>>>> By keeping memcg structures hanging around until the last referring kmem
>>>> page is uncharged do such zombie memcg each consume a css_id and thus
>>>> put pressure on the 64k css_id space? I imagine in pathological cases
>>>> this would prevent creation of new cgroups until these zombies are
>>>> dereferenced.
>>>
>>> Yes, but although this patch makes it more likely, it doesn't introduce
>>> that. If the tasks, for instance, grab a reference to the cgroup dentry
>>> in the filesystem (like their CWD, etc), they will also keep the cgroup
>>> around.
>>
>> Fair point. But this doesn't seems like a feature. It's probably not
>> needed initially, but what do you think about creating a
>> memcg_kernel_context structure which is allocated when memcg is
>> allocated? Kernel pages charged to a memcg would have
>> page_cgroup->mem_cgroup=memcg_kernel_context rather than memcg. This
>> would allow the mem_cgroup and its css_id to be deleted when the cgroup
>> is unlinked from cgroupfs while allowing for the active kernel pages to
>> continue pointing to a valid memcg_kernel_context. This would be a
>> reference counted structure much like you are doing with memcg. When a
>> memcg is deleted the memcg_kernel_context would be linked into its
>> surviving parent memcg. This would avoid needing to visit each kernel
>> page.
>
> You need more, you need at the res_counters to stay around as well. And
> probably other fields.

I am not sure the res_counters would need to stay around. Once a memcg_kernel_context has been reparented, then any future kernel page uncharge calls will uncharge the parent res_counter.

> So my fear here is that as you add fields to that structure, you can
> defeat a bit the goal of reducing memory consumption. Still leaves the
> css space, yes. But by doing this we can introduce some subtle bugs by
> having a field in the wrong structure.
>
> Did you observe that to be a big problem in your systems?

No I have not seen this yet. But our past solutions have reparented kmem_cache's to root memcg so we have been avoiding zombie memcg. My concerns with your approach are just a suspicion because we have been experimenting with accounting of even more kernel memory (e.g. vmalloc, kernel stacks, page tables). As the scope of such accounting grows the chance of long lived charged pages grows and thus the chance of zombies which exhaust the css_id space grows.

Subject: Re: [PATCH v2 04/11] kmem accounting basic infrastructure
Posted by Christoph Lameter on Wed, 15 Aug 2012 17:26:25 GMT
View Forum Message <> Reply to Message

On Wed, 15 Aug 2012, Glauber Costa wrote:

> Remember we copy over the metadata and create copies of the caches
> per-memcg. Therefore, a dentry belongs to a memcg if it was allocated
> from the slab pertaining to that memcg.

The dentry could be used by other processes in the system though. F.e. directory names could easily be created by one process and then used by a multitude of others.

> It is not 100 % accurate, but it is good enough.

Lets hope that is true.

Subject: Re: [PATCH v2 04/11] kmem accounting basic infrastructure
Posted by Glauber Costa on Wed, 15 Aug 2012 18:00:43 GMT
View Forum Message <> Reply to Message

On 08/15/2012 10:01 PM, Ying Han wrote:
> On Wed, Aug 15, 2012 at 5:39 AM, Michal Hocko <mhocko@suse.cz> wrote:
>> On Wed 15-08-12 13:33:55, Glauber Costa wrote:

>> [...]
>>>> This can
>>>> be quite confusing. I am still not sure whether we should mix the two
>>>> things together. If somebody wants to limit the kernel memory he has to
>>>> touch the other limit anyway. Do you have a strong reason to mix the
>>>> user and kernel counters?
>>>
>>> This is funny, because the first opposition I found to this work was
>>> "Why would anyone want to limit it separately?" =p
>>>
>>> It seems that a quite common use case is to have a container with a
>>> unified view of "memory" that it can use the way he likes, be it with
>>> kernel memory, or user memory. I believe those people would be happy to
>>> just silently account kernel memory to user memory, or at the most have
>>> a switch to enable it.
>>>
>>> What gets clear from this back and forth, is that there are people
>>> interested in both use cases.
>>
>> I am still not 100% sure myself. It is just clear that the reclaim would
>> need some work in order to do accounting like this.
>>
>>>> My impression was that kernel allocation should simply fail while user
>>>> allocations might reclaim as well. Why should we reclaim just because of
>>>> the kernel allocation (which is unreclaimable from hard limit reclaim
>>>> point of view)?
>>>
>>> That is not what the kernel does, in general. We assume that if he wants
>>> that memory and we can serve it, we should. Also, not all kernel memory
>>> is unreclaimable. We can shrink the slabs, for instance. Ying Han
>>> claims she has patches for that already...
>>
>> Are those patches somewhere around?
>
> Yes, I am working on it to post it sometime *this week*. My last
> rebase is based on v3.3 and now I am trying to get it rebased to
> github-memcg. The patch itself has a functional dependency on kernel
> slab accounting, and I am trying to get that rebased on Glauber's tree
> but has some difficulty now. What I am planning to do is post the RFC
> w/ only complied version by far.

That would be great, so we can start looking at its design, at least.

> The patch handles dentry cache shrinker only at this moment. That is
> what we discussed last time as well, where dentry contributes most of
> the reclaimable objects. (it pins inode, so we leave inode behind)
>

This will mark the inodes as reclaimable, but will leave them in memory.
If we are assuming memory pressure, it would be good to shrink them too.

Subject: Re: [PATCH v2 04/11] kmem accounting basic infrastructure
Posted by Ying Han on Wed, 15 Aug 2012 18:01:41 GMT
View Forum Message <> Reply to Message

On Wed, Aug 15, 2012 at 5:39 AM, Michal Hocko <mhocko@suse.cz> wrote:
> On Wed 15-08-12 13:33:55, Glauber Costa wrote:
> [...]
>> > This can
>> > be quite confusing. I am still not sure whether we should mix the two
>> > things together. If somebody wants to limit the kernel memory he has to
>> > touch the other limit anyway. Do you have a strong reason to mix the
>> > user and kernel counters?
>>
>> This is funny, because the first opposition I found to this work was
>> "Why would anyone want to limit it separately?" =p
>>
>> It seems that a quite common use case is to have a container with a
>> unified view of "memory" that it can use the way he likes, be it with
>> kernel memory, or user memory. I believe those people would be happy to
>> just silently account kernel memory to user memory, or at the most have
>> a switch to enable it.
>>
>> What gets clear from this back and forth, is that there are people
>> interested in both use cases.
>
> I am still not 100% sure myself. It is just clear that the reclaim would
> need some work in order to do accounting like this.
>
>> > My impression was that kernel allocation should simply fail while user
>> > allocations might reclaim as well. Why should we reclaim just because of
>> > the kernel allocation (which is unreclaimable from hard limit reclaim
>> > point of view)?
>>
>> That is not what the kernel does, in general. We assume that if he wants
>> that memory and we can serve it, we should. Also, not all kernel memory
>> is unreclaimable. We can shrink the slabs, for instance. Ying Han
>> claims she has patches for that already...
>
> Are those patches somewhere around?

Yes, I am working on it to post it sometime *this week*. My last
rebase is based on v3.3 and now I am trying to get it rebased to
github-memcg. The patch itself has a functional dependency on kernel
slab accounting, and I am trying to get that rebased on Glauber's tree

but has some difficulty now. What I am planning to do is post the RFC
w/ only complied version by far.

The patch handles dentry cache shrinker only at this moment. That is
what we discussed last time as well, where dentry contributes most of
the reclaimable objects. (it pins inode, so we leave inode behind)

--Ying
>
> [...]
>> > This doesn't check for the hierachy so kmem_accounted might not be in
>> > sync with it's parents. mem_cgroup_create (below) needs to copy
>> > kmem_accounted down from the parent and the above needs to check if this
>> > is a similar dance like mem_cgroup_oom_control_write.
>> >
>>
>> I don't see why we have to.
>>
>> I believe in a A/B/C hierarchy, C should be perfectly able to set a
>> different limit than its parents. Note that this is not a boolean.
>
> Ohh, I wasn't clear enough. I am not against setting the _limit_ I just
> meant that the kmem_accounted should be consistent within the hierarchy.
>
> --
> Michal Hocko
> SUSE Labs
>
> --
> 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

Subject: Re: [PATCH v2 04/11] kmem accounting basic infrastructure
Posted by Ying Han on Wed, 15 Aug 2012 18:07:09 GMT
View Forum Message <> Reply to Message

On Wed, Aug 15, 2012 at 8:11 AM, Glauber Costa <glommer@parallels.com> wrote:
> On 08/15/2012 06:47 PM, Christoph Lameter wrote:
>> On Wed, 15 Aug 2012, Michal Hocko wrote:
>>
>>>> That is not what the kernel does, in general. We assume that if he wants
>>>> that memory and we can serve it, we should. Also, not all kernel memory
>>>> is unreclaimable. We can shrink the slabs, for instance. Ying Han
>>>> claims she has patches for that already...

>>>
>>> Are those patches somewhere around?
>>
>> You can already shrink the reclaimable slabs (dentries / inodes) via
>> calls to the subsystem specific shrinkers. Did Ying Han do anything to
>> go beyond that?
>>
> That is not enough for us.
> We would like to make sure that the objects being discarded belong to
> the memcg which is under pressure. We don't need to be perfect here, and
> an occasional slip is totally fine. But if in general, shrinking from
> memcg A will mostly wipe out objects from memcg B, we harmed the system
> in return for nothing good.

Correct. For example, we have per-superblock shrinker today for vfs
caches. That is not enough since we need to isolate the dentry caches
per-memcg basis.

--Ying
>
> --
> 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

Subject: Re: [PATCH v2 04/11] kmem accounting basic infrastructure
Posted by Ying Han on Wed, 15 Aug 2012 18:11:49 GMT
View Forum Message <> Reply to Message

On Wed, Aug 15, 2012 at 8:34 AM, Christoph Lameter <cl@linux.com> wrote:
> On Wed, 15 Aug 2012, Glauber Costa wrote:
>
>> On 08/15/2012 06:47 PM, Christoph Lameter wrote:
>> > On Wed, 15 Aug 2012, Michal Hocko wrote:
>> >
>> >> That is not what the kernel does, in general. We assume that if he wants
>> >> that memory and we can serve it, we should. Also, not all kernel memory
>> >> is unreclaimable. We can shrink the slabs, for instance. Ying Han
>> >> claims she has patches for that already...
>> >>
>> >> Are those patches somewhere around?
>> >
>> > You can already shrink the reclaimable slabs (dentries / inodes) via
>> > calls to the subsystem specific shrinkers. Did Ying Han do anything to
>> > go beyond that?
>> >

>> That is not enough for us.
>> We would like to make sure that the objects being discarded belong to
>> the memcg which is under pressure. We don't need to be perfect here, and
>> an occasional slip is totally fine. But if in general, shrinking from
>> memcg A will mostly wipe out objects from memcg B, we harmed the system
>> in return for nothing good.
>
> How can you figure out which objects belong to which memcg? The ownerships
> of dentries and inodes is a dubious concept already.

I figured it out based on the kernel slab accounting.
obj->page->kmem_cache->memcg

--Ying
>
> --
> 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

Subject: Re: [PATCH v2 04/11] kmem accounting basic infrastructure
Posted by Christoph Lameter on Wed, 15 Aug 2012 18:25:04 GMT
View Forum Message <> Reply to Message

On Wed, 15 Aug 2012, Ying Han wrote:

> > How can you figure out which objects belong to which memcg? The ownerships
> > of dentries and inodes is a dubious concept already.
>
> I figured it out based on the kernel slab accounting.
> obj->page->kmem_cache->memcg

Well that is only the memcg which allocated it. It may be in use heavily
by other processes.

Subject: Re: [PATCH v2 04/11] kmem accounting basic infrastructure
Posted by Glauber Costa on Wed, 15 Aug 2012 19:22:34 GMT
View Forum Message <> Reply to Message

On 08/15/2012 10:25 PM, Christoph Lameter wrote:
> On Wed, 15 Aug 2012, Ying Han wrote:
>
>>> How can you figure out which objects belong to which memcg? The ownerships
>>> of dentries and inodes is a dubious concept already.

>>
>> I figured it out based on the kernel slab accounting.
>> obj->page->kmem_cache->memcg
>
> Well that is only the memcg which allocated it. It may be in use heavily
> by other processes.
>

Yes, but a lot of the use cases for cgroups/containers are pretty local.
That is why we have been able to get away with a first-touch mechanism
even in user pages memcg. In those cases - which we expect to be the
majority of them - this will perform well.

Now, this is not of course representative of the whole range of possible
use cases, and others are valid. There are people like Greg
and Ying Han herself that want a more fine grained control on which
memcg gets the accounting. That is one of the topics for the summit.

But even then: regardless of what mechanism is in place, one cgroup is
to be accounted (or not accounted at all, meaning it belongs to a
non-accounted cgroup). And then we can just grab whichever memcg it was
allocated from and shrink it.

Subject: Re: [PATCH v2 06/11] memcg: kmem controller infrastructure
Posted by Glauber Costa on Wed, 15 Aug 2012 19:31:34 GMT
View Forum Message <> Reply to Message

On 08/15/2012 09:12 PM, Greg Thelen wrote:
> On Wed, Aug 15 2012, Glauber Costa wrote:
>
>> On 08/15/2012 08:38 PM, Greg Thelen wrote:
>>> On Wed, Aug 15 2012, Glauber Costa wrote:
>>>
>>>> On 08/14/2012 10:58 PM, Greg Thelen wrote:
>>>>> On Mon, Aug 13 2012, Glauber Costa wrote:
>>>>>
>>>>>> + WARN_ON(mem_cgroup_is_root(memcg));
>>>>>> + size = (1 << order) << PAGE_SHIFT;
>>>>>> + memcg_uncharge_kmem(memcg, size);
>>>>>> + mem_cgroup_put(memcg);
>>>>> Why do we need ref-counting here ? kmem res_counter cannot work as
>>>>> reference ?
>>>> This is of course the pair of the mem_cgroup_get() you commented on
>>>> earlier. If we need one, we need the other. If we don't need one, we
>>>> don't need the other =)
>>>>
>>>> The guarantee we're trying to give here is that the memcg structure will

>>>>>> stay around while there are dangling charges to kmem, that we decided
>>>>>> not to move (remember: moving it for the stack is simple, for the slab
>>>>>> is very complicated and ill-defined, and I believe it is better to treat
>>>>>> all kmem equally here)
>>>>>
>>>>> By keeping memcg structures hanging around until the last referring kmem
>>>>> page is uncharged do such zombie memcg each consume a css_id and thus
>>>>> put pressure on the 64k css_id space? I imagine in pathological cases
>>>>> this would prevent creation of new cgroups until these zombies are
>>>>> dereferenced.
>>>>
>>>> Yes, but although this patch makes it more likely, it doesn't introduce
>>>> that. If the tasks, for instance, grab a reference to the cgroup dentry
>>>> in the filesystem (like their CWD, etc), they will also keep the cgroup
>>>> around.
>>>
>>> Fair point. But this doesn't seems like a feature. It's probably not
>>> needed initially, but what do you think about creating a
>>> memcg_kernel_context structure which is allocated when memcg is
>>> allocated? Kernel pages charged to a memcg would have
>>> page_cgroup->mem_cgroup=memcg_kernel_context rather than memcg. This
>>> would allow the mem_cgroup and its css_id to be deleted when the cgroup
>>> is unlinked from cgroupfs while allowing for the active kernel pages to
>>> continue pointing to a valid memcg_kernel_context. This would be a
>>> reference counted structure much like you are doing with memcg. When a
>>> memcg is deleted the memcg_kernel_context would be linked into its
>>> surviving parent memcg. This would avoid needing to visit each kernel
>>> page.
>>
>> You need more, you need at the res_counters to stay around as well. And
>> probably other fields.
>
> I am not sure the res_counters would need to stay around. Once a
> memcg_kernel_context has been reparented, then any future kernel page
> uncharge calls will uncharge the parent res_counter.

Well, if you hold the memcg due to a reference, like in the dentry case,
then fine. But if this is a dangling charge, as will be the case with
the slab, then you have to uncharge it.

An arbitrary number of parents might have been deleted as well, so you
need to transverse them all until you reach a live parent to uncharge from.

To do that, your counters have to be still alive.

>
>> So my fear here is that as you add fields to that structure, you can
>> defeat a bit the goal of reducing memory consumption. Still leaves the

>> css space, yes. But by doing this we can introduce some subtle bugs by
>> having a field in the wrong structure.
>>
>> Did you observe that to be a big problem in your systems?
>
> No I have not seen this yet. But our past solutions have reparented
> kmem_cache's to root memcg so we have been avoiding zombie memcg. My
> concerns with your approach are just a suspicion because we have been
> experimenting with accounting of even more kernel memory (e.g. vmalloc,
> kernel stacks, page tables). As the scope of such accounting grows the
> chance of long lived charged pages grows and thus the chance of zombies
> which exhaust the css_id space grows.

Well, since we agree this can all be done under the hood, I'd say let's
wait until a problem actually exists, since the solution is likely to be
a bit convoluted...

I personally believe that if won't have a lot of task movement, most of
the data will go away as the cgroup dies. The remainder shouldn't be too
much to hold it in memory for a lot of time. This is of course assuming
a real use case, not an adversarial scenario, which is quite easy to
come up with: just create a task, hold a bunch of kmem, move the task
away, delete the cgroup, etc.

That said, nothing stops us to actively try to create a scenario that
would demonstrate such a problem.

Subject: Re: [PATCH v2 04/11] kmem accounting basic infrastructure
Posted by Ying Han on Wed, 15 Aug 2012 19:50:55 GMT
View Forum Message <> Reply to Message

On Tue, Aug 14, 2012 at 9:21 AM, Michal Hocko <mhocko@suse.cz> wrote:
> On Thu 09-08-12 17:01:12, Glauber Costa wrote:
>> This patch adds the basic infrastructure for the accounting of the slab
>> caches. 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.
>>
>> The code is not enabled until a limit is set. This can be tested by the
>> flag "kmem_accounted". This means that after the patch is applied, no
>> behavioral changes exists for whoever is still using memcg to control

>> their memory usage.
>>
>> 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.
>
> Well, it contributes to the user limit so it is not unlimited. It just
> falls under a different limit and it tends to contribute less. This can
> be quite confusing. I am still not sure whether we should mix the two
> things together. If somebody wants to limit the kernel memory he has to
> touch the other limit anyway. Do you have a strong reason to mix the
> user and kernel counters?

The reason to mix the two together is a compromise of the two use
cases we've heard by far. In google, we only need one limit which
limits u & k, and the reclaim kicks in when the total usage hits the
limit.

> My impression was that kernel allocation should simply fail while user
> allocations might reclaim as well. Why should we reclaim just because of
> the kernel allocation (which is unreclaimable from hard limit reclaim
> point of view)?

Some of kernel objects are reclaimable if we have per-memcg shrinker.

> I also think that the whole thing would get much simpler if those two
> are split. Anyway if this is really a must then this should be
> documented here.

What would be the use case you have in your end?

--Ying

> One nit bellow.
>
>> 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)
>>
>> Signed-off-by: Glauber Costa <glommer@parallels.com>
>> CC: Michal Hocko <mhocko@suse.cz>
>> CC: Johannes Weiner <hannes@cmpxchg.org>
>> Reviewed-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
>> ---
>> mm/memcontrol.c | 69
++-

```
>>  1 file changed, 68 insertions(+), 1 deletion(-)
>>
>> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
>> index b0e29f4..54e93de 100644
>> --- a/mm/memcontrol.c
>> +++ b/mm/memcontrol.c
> [...]
>> @@ -4046,8 +4059,23 @@ 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 = res_counter_set_limit(&memcg->kmem, val);
>> +                if (ret)
>> +                        break;
>> +                /*
>> +                 * Once enabled, can't be disabled. We could in theory
>> +                 * disable it if we haven't yet created any caches, or
>> +                 * if we can shrink them all to death.
>> +                 *
>> +                 * But it is not worth the trouble
>> +                 */
>> +                if (!memcg->kmem_accounted && val != RESOURCE_MAX)
>> +                        memcg->kmem_accounted = true;
>> +        } else
>> +                return -EINVAL;
>>          break;
>
> This doesn't check for the hierachy so kmem_accounted might not be in
> sync with it's parents. mem_cgroup_create (below) needs to copy
> kmem_accounted down from the parent and the above needs to check if this
> is a similar dance like mem_cgroup_oom_control_write.
>
> [...]
>
>> @@ -5033,6 +5098,7 @@ 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.
>> @@ -5043,6 +5109,7 @@ mem_cgroup_create(struct cgroup *cont)
>>      } else {
```

>> res_counter_init(&memcg->res, NULL);
>> res_counter_init(&memcg->memsw, NULL);
>> + res_counter_init(&memcg->kmem, NULL);
>> }
>> memcg->last_scanned_node = MAX_NUMNODES;
>> INIT_LIST_HEAD(&memcg->oom_notify);
>> --
>> 1.7.11.2
>>
>> --
>> To unsubscribe from this list: send the line "unsubscribe cgroups" in
>> the body of a message to majordomo@vger.kernel.org
>> More majordomo info at http://vger.kernel.org/majordomo-info.html
>
> --
> Michal Hocko
> SUSE Labs
>
> --
> 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

Subject: Re: [PATCH v2 06/11] memcg: kmem controller infrastructure
Posted by Greg Thelen on Thu, 16 Aug 2012 03:37:01 GMT
View Forum Message <> Reply to Message

On Wed, Aug 15 2012, Glauber Costa wrote:

> On 08/15/2012 09:12 PM, Greg Thelen wrote:
>> On Wed, Aug 15 2012, Glauber Costa wrote:
>>
>>> On 08/15/2012 08:38 PM, Greg Thelen wrote:
>>>> On Wed, Aug 15 2012, Glauber Costa wrote:
>>>>
>>>>> On 08/14/2012 10:58 PM, Greg Thelen wrote:
>>>>>> On Mon, Aug 13 2012, Glauber Costa wrote:
>>>>>>
>>>>>>>> + WARN_ON(mem_cgroup_is_root(memcg));
>>>>>>>> + size = (1 << order) << PAGE_SHIFT;
>>>>>>>> + memcg_uncharge_kmem(memcg, size);
>>>>>>>> + mem_cgroup_put(memcg);
>>>>>>> Why do we need ref-counting here ? kmem res_counter cannot work as
>>>>>>> reference ?
>>>>>> This is of course the pair of the mem_cgroup_get() you commented on
>>>>>> earlier. If we need one, we need the other. If we don't need one, we

>>>>>>> don't need the other =)
>>>>>>>
>>>>>>> The guarantee we're trying to give here is that the memcg structure will
>>>>>>> stay around while there are dangling charges to kmem, that we decided
>>>>>>> not to move (remember: moving it for the stack is simple, for the slab
>>>>>>> is very complicated and ill-defined, and I believe it is better to treat
>>>>>>> all kmem equally here)
>>>>>>
>>>>>> By keeping memcg structures hanging around until the last referring kmem
>>>>>> page is uncharged do such zombie memcg each consume a css_id and thus
>>>>>> put pressure on the 64k css_id space? I imagine in pathological cases
>>>>>> this would prevent creation of new cgroups until these zombies are
>>>>>> dereferenced.
>>>>>
>>>>> Yes, but although this patch makes it more likely, it doesn't introduce
>>>>> that. If the tasks, for instance, grab a reference to the cgroup dentry
>>>>> in the filesystem (like their CWD, etc), they will also keep the cgroup
>>>>> around.
>>>>
>>>> Fair point. But this doesn't seems like a feature. It's probably not
>>>> needed initially, but what do you think about creating a
>>>> memcg_kernel_context structure which is allocated when memcg is
>>>> allocated? Kernel pages charged to a memcg would have
>>>> page_cgroup->mem_cgroup=memcg_kernel_context rather than memcg. This
>>>> would allow the mem_cgroup and its css_id to be deleted when the cgroup
>>>> is unlinked from cgroupfs while allowing for the active kernel pages to
>>>> continue pointing to a valid memcg_kernel_context. This would be a
>>>> reference counted structure much like you are doing with memcg. When a
>>>> memcg is deleted the memcg_kernel_context would be linked into its
>>>> surviving parent memcg. This would avoid needing to visit each kernel
>>>> page.
>>>
>>> You need more, you need at the res_counters to stay around as well. And
>>> probably other fields.
>>
>> I am not sure the res_counters would need to stay around. Once a
>> memcg_kernel_context has been reparented, then any future kernel page
>> uncharge calls will uncharge the parent res_counter.
>
> Well, if you hold the memcg due to a reference, like in the dentry case,
> then fine. But if this is a dangling charge, as will be the case with
> the slab, then you have to uncharge it.
>
> An arbitrary number of parents might have been deleted as well, so you
> need to transverse them all until you reach a live parent to uncharge from.

I was thinking that each time a memcg is deleted move the
memcg_kernel_context from the victim memcg to its parent. When moving,

also update the context to refer to the parent and link context to
parent:
```
 for_each_kernel_context(kernel_context, memcg) {
   kernel_context->memcg = memcg->parent;
   list_add(&kernel_context->list, &memcg->parent->kernel_contexts);
 }
```

Whenever pages referring to a memcg_kernel_context are uncharged they
will uncharge the nearest surviving parent memcg.

> To do that, your counters have to be still alive.

The counters of nearest surviving parent will be alive and pointed to by
memcg_kernel_context->memcg.

>>> So my fear here is that as you add fields to that structure, you can
>>> defeat a bit the goal of reducing memory consumption. Still leaves the
>>> css space, yes. But by doing this we can introduce some subtle bugs by
>>> having a field in the wrong structure.
>>>
>>> Did you observe that to be a big problem in your systems?
>>
>> No I have not seen this yet. But our past solutions have reparented
>> kmem_cache's to root memcg so we have been avoiding zombie memcg. My
>> concerns with your approach are just a suspicion because we have been
>> experimenting with accounting of even more kernel memory (e.g. vmalloc,
>> kernel stacks, page tables). As the scope of such accounting grows the
>> chance of long lived charged pages grows and thus the chance of zombies
>> which exhaust the css_id space grows.
>
> Well, since we agree this can all be done under the hood, I'd say let's
> wait until a problem actually exists, since the solution is likely to be
> a bit convoluted...
>
> I personally believe that if won't have a lot of task movement, most of
> the data will go away as the cgroup dies. The remainder shouldn't be too
> much to hold it in memory for a lot of time. This is of course assuming
> a real use case, not an adversarial scenario, which is quite easy to
> come up with: just create a task, hold a bunch of kmem, move the task
> away, delete the cgroup, etc.
>
> That said, nothing stops us to actively try to create a scenario that
> would demonstrate such a problem.

With our in-house per-memcg slab accounting (similar to what's discussed
here), we're seeing a few slab allocations (mostly radix_tree_node) that
survive a long time after memcg deletion. This isn't meant as criticism
of this patch series, just an fyi that I expect there will be scenarios

where some dead kmem caches will live for a long time. Though I think
that in your patches a dead kmem cache does not hold reference to the
memcg.

Subject: Re: [PATCH v2 06/11] memcg: kmem controller infrastructure
Posted by Glauber Costa on Thu, 16 Aug 2012 07:47:43 GMT
View Forum Message <> Reply to Message

On 08/16/2012 07:37 AM, Greg Thelen wrote:
> On Wed, Aug 15 2012, Glauber Costa wrote:
>
>> On 08/15/2012 09:12 PM, Greg Thelen wrote:
>>> On Wed, Aug 15 2012, Glauber Costa wrote:
>>>
>>>> On 08/15/2012 08:38 PM, Greg Thelen wrote:
>>>>> On Wed, Aug 15 2012, Glauber Costa wrote:
>>>>>
>>>>>> On 08/14/2012 10:58 PM, Greg Thelen wrote:
>>>>>>> On Mon, Aug 13 2012, Glauber Costa wrote:
>>>>>>>
>>>>>>>>>>>>> + WARN_ON(mem_cgroup_is_root(memcg));
>>>>>>>>>>>>> + size = (1 << order) << PAGE_SHIFT;
>>>>>>>>>>>>> + memcg_uncharge_kmem(memcg, size);
>>>>>>>>>>>>> + mem_cgroup_put(memcg);
>>>>>>>>>>> Why do we need ref-counting here ? kmem res_counter cannot work as
>>>>>>>>>>> reference ?
>>>>>>>>>> This is of course the pair of the mem_cgroup_get() you commented on
>>>>>>>>>> earlier. If we need one, we need the other. If we don't need one, we
>>>>>>>>>> don't need the other =)
>>>>>>>>>>
>>>>>>>>>> The guarantee we're trying to give here is that the memcg structure will
>>>>>>>>>> stay around while there are dangling charges to kmem, that we decided
>>>>>>>>>> not to move (remember: moving it for the stack is simple, for the slab
>>>>>>>>>> is very complicated and ill-defined, and I believe it is better to treat
>>>>>>>>>> all kmem equally here)
>>>>>>>>>>
>>>>>>>>> By keeping memcg structures hanging around until the last referring kmem
>>>>>>>>> page is uncharged do such zombie memcg each consume a css_id and thus
>>>>>>>>> put pressure on the 64k css_id space? I imagine in pathological cases
>>>>>>>>> this would prevent creation of new cgroups until these zombies are
>>>>>>>>> dereferenced.
>>>>>>>>>
>>>>>>>> Yes, but although this patch makes it more likely, it doesn't introduce
>>>>>>>> that. If the tasks, for instance, grab a reference to the cgroup dentry
>>>>>>>> in the filesystem (like their CWD, etc), they will also keep the cgroup
>>>>>>>> around.
>>>>>>>

>>>>> Fair point. But this doesn't seems like a feature. It's probably not
>>>>> needed initially, but what do you think about creating a
>>>>> memcg_kernel_context structure which is allocated when memcg is
>>>>> allocated? Kernel pages charged to a memcg would have
>>>>> page_cgroup->mem_cgroup=memcg_kernel_context rather than memcg. This
>>>>> would allow the mem_cgroup and its css_id to be deleted when the cgroup
>>>>> is unlinked from cgroupfs while allowing for the active kernel pages to
>>>>> continue pointing to a valid memcg_kernel_context. This would be a
>>>>> reference counted structure much like you are doing with memcg. When a
>>>>> memcg is deleted the memcg_kernel_context would be linked into its
>>>>> surviving parent memcg. This would avoid needing to visit each kernel
>>>>> page.
>>>>
>>>> You need more, you need at the res_counters to stay around as well. And
>>>> probably other fields.
>>>
>>> I am not sure the res_counters would need to stay around. Once a
>>> memcg_kernel_context has been reparented, then any future kernel page
>>> uncharge calls will uncharge the parent res_counter.
>>
>> Well, if you hold the memcg due to a reference, like in the dentry case,
>> then fine. But if this is a dangling charge, as will be the case with
>> the slab, then you have to uncharge it.
>>
>> An arbitrary number of parents might have been deleted as well, so you
>> need to transverse them all until you reach a live parent to uncharge from.
>
> I was thinking that each time a memcg is deleted move the
> memcg_kernel_context from the victim memcg to its parent. When moving,
> also update the context to refer to the parent and link context to
> parent:
>   for_each_kernel_context(kernel_context, memcg) {
>   kernel_context->memcg = memcg->parent;
>   list_add(&kernel_context->list, &memcg->parent->kernel_contexts);
>   }
>
> Whenever pages referring to a memcg_kernel_context are uncharged they
> will uncharge the nearest surviving parent memcg.
>
>> To do that, your counters have to be still alive.
>
> The counters of nearest surviving parent will be alive and pointed to by
> memcg_kernel_context->memcg.
>
>>>> So my fear here is that as you add fields to that structure, you can
>>>> defeat a bit the goal of reducing memory consumption. Still leaves the
>>>> css space, yes. But by doing this we can introduce some subtle bugs by
>>>> having a field in the wrong structure.

>>>>
>>>> Did you observe that to be a big problem in your systems?
>>>
>>> No I have not seen this yet. But our past solutions have reparented
>>> kmem_cache's to root memcg so we have been avoiding zombie memcg. My
>>> concerns with your approach are just a suspicion because we have been
>>> experimenting with accounting of even more kernel memory (e.g. vmalloc,
>>> kernel stacks, page tables). As the scope of such accounting grows the
>>> chance of long lived charged pages grows and thus the chance of zombies
>>> which exhaust the css_id space grows.
>>

Can't we just free the css_id, and convention that it should not be used
after mem_cgroup_destroy()? The memory will still stay around,
sure, but at least the pressure on the css_id space goes away.

I am testing a patch that does precisely that here, and will let you
know of the results. But if you were willing to have a smaller structure
just to serve as a zombie, any approach that works for it would have to
assume the css_id was already freed, so I don't anticipate huge problems.

>> Well, since we agree this can all be done under the hood, I'd say let's
>> wait until a problem actually exists, since the solution is likely to be
>> a bit convoluted...
>>
>> I personally believe that if won't have a lot of task movement, most of
>> the data will go away as the cgroup dies. The remainder shouldn't be too
>> much to hold it in memory for a lot of time. This is of course assuming
>> a real use case, not an adversarial scenario, which is quite easy to
>> come up with: just create a task, hold a bunch of kmem, move the task
>> away, delete the cgroup, etc.
>>
>> That said, nothing stops us to actively try to create a scenario that
>> would demonstrate such a problem.
>
> With our in-house per-memcg slab accounting (similar to what's discussed
> here), we're seeing a few slab allocations (mostly radix_tree_node) that
> survive a long time after memcg deletion. This isn't meant as criticism
> of this patch series, just an fyi that I expect there will be scenarios
> where some dead kmem caches will live for a long time. Though I think
> that in your patches a dead kmem cache does not hold reference to the
> memcg.
>

Does shrinking help?

One of the things I was thinking about doing when we have proper
per-memcg shrinking, is to shrink all caches when destroying the memcg.

Because the memcg is dead, we'll have no more memcg pressure, and those
will go away only when global pressure comes to play. Which means that
the references will then be around for a very long time. What is the
best behavior is debatable, but at least at first, I'd stand by the side
of getting rid of everything the memcg created as much as possible.

Also, if you are concerned with memory usage due to the memcg structure,
bear in mind that the caches metadata may be considerably more...

Subject: Re: [PATCH v2 06/11] memcg: kmem controller infrastructure
Posted by Michal Hocko on Thu, 16 Aug 2012 09:53:09 GMT
View Forum Message <> Reply to Message

On Wed 15-08-12 18:27:45, Glauber Costa wrote:
>
> >>
> >> I see now, you seem to be right.
> >
> > No I am not because it seems that I am really blind these days...
> > We were doing this in mem_cgroup_do_charge for ages:
> > if (!(gfp_mask & __GFP_WAIT))
> > return CHARGE_WOULDBLOCK;
> >
> > /me goes to hide and get with further feedback with a clean head.
> >
> > Sorry about that.
> >
> I am as well, since I went to look at mem_cgroup_do_charge() and missed
> that.

I thought we are not doing atomic allocations in user pages accounting
but I was obviously wrong because at least shmem uses atomic
allocations for ages.

> Do you have any other concerns specific to this patch ?

I understood you changed also handle thingy. So the patch should be
correct.
Do you plan to send an updated version?

--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v2 06/11] memcg: kmem controller infrastructure
Posted by Glauber Costa on Thu, 16 Aug 2012 09:57:07 GMT
View Forum Message <> Reply to Message

On 08/16/2012 01:53 PM, Michal Hocko wrote:
> On Wed 15-08-12 18:27:45, Glauber Costa wrote:
>>
>>>>
>>>> I see now, you seem to be right.
>>>
>>> No I am not because it seems that I am really blind these days...
>>> We were doing this in mem_cgroup_do_charge for ages:
>>> if (!(gfp_mask & __GFP_WAIT))
>>> return CHARGE_WOULDBLOCK;
>>>
>>> /me goes to hide and get with further feedback with a clean head.
>>>
>>> Sorry about that.
>>>
>> I am as well, since I went to look at mem_cgroup_do_charge() and missed
>> that.
>
> I thought we are not doing atomic allocations in user pages accounting
> but I was obviously wrong because at least shmem uses atomic
> allocations for ages.
>
>> Do you have any other concerns specific to this patch ?
>
> I understood you changed also handle thingy. So the patch should be
> correct.
> Do you plan to send an updated version?
>
That depends more on you than on me! =)

Do you still have any concerns regarding the u+k charging as it stands
now? That would be the last big concern I heard during this iteration.

If you are happy with the answers you got so far, and believe it is
acceptable to proceed with the charging this way, I will be ready to
send an updated version soon.

Subject: Re: [PATCH v2 06/11] memcg: kmem controller infrastructure
Posted by Michal Hocko on Thu, 16 Aug 2012 15:05:18 GMT
View Forum Message <> Reply to Message

On Thu 16-08-12 13:57:07, Glauber Costa wrote:
> On 08/16/2012 01:53 PM, Michal Hocko wrote:

> > On Wed 15-08-12 18:27:45, Glauber Costa wrote:
> >>
> >>>>
> >>>> I see now, you seem to be right.
> >>>
> >>> No I am not because it seems that I am really blind these days...
> >>> We were doing this in mem_cgroup_do_charge for ages:
> >>> if (!(gfp_mask & __GFP_WAIT))
> >>> return CHARGE_WOULDBLOCK;
> >>>
> >>> /me goes to hide and get with further feedback with a clean head.
> >>>
> >>> Sorry about that.
> >>>
> >> I am as well, since I went to look at mem_cgroup_do_charge() and missed
> >> that.
> >
> > I thought we are not doing atomic allocations in user pages accounting
> > but I was obviously wrong because at least shmem uses atomic
> > allocations for ages.
> >
> >> Do you have any other concerns specific to this patch ?
> >
> > I understood you changed also handle thingy. So the patch should be
> > correct.
> > Do you plan to send an updated version?
> >
> That depends more on you than on me! =)
>
> Do you still have any concerns regarding the u+k charging as it stands
> now? That would be the last big concern I heard during this iteration.

Well, I am still not 100% sure because I still see technical
difficulties that are not addressed by the patchset (memcg-oom, memcg
slab shrinking, possibly others). More importantly this is changing the
current semantic of the limit so we should better be careful about it
and check that we are not making the code tight to specific workloads
without a way out.

On the other hand I do not want to block the progress here without
having _really_ good arguments against that couldn't be handled later
(and it seems that some of my concerns are work in progress already).

I have to admit I like several things about the patchset. Especially the
way how it enables easy-to-setup (aka don't care about kmem details just
make sure you can cap the thing) as well as "I know exactly what I want
to do" usecases.
It is also good nice that only users of the feature are affected by

potential issues.

So I think it is worth a broader attention which could produce other use cases which could show potential drawbacks from the u+k semantic but I would be still very careful about merging it to the Linus tree and only merge it after at least the memcg reclaim path is slab aware. Living in the -mm tree should help us with the testing converage.

Does it sounds reasonable?
--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v2 06/11] memcg: kmem controller infrastructure
Posted by Glauber Costa on Thu, 16 Aug 2012 15:22:45 GMT
View Forum Message <> Reply to Message

On 08/16/2012 07:05 PM, Michal Hocko wrote:
> On Thu 16-08-12 13:57:07, Glauber Costa wrote:
>> On 08/16/2012 01:53 PM, Michal Hocko wrote:
>>> On Wed 15-08-12 18:27:45, Glauber Costa wrote:
>>>>
>>>>>>
>>>>>> I see now, you seem to be right.
>>>>>
>>>>> No I am not because it seems that I am really blind these days...
>>>>> We were doing this in mem_cgroup_do_charge for ages:
>>>>> if (!(gfp_mask & __GFP_WAIT))
>>>>> return CHARGE_WOULDBLOCK;
>>>>>
>>>>> /me goes to hide and get with further feedback with a clean head.
>>>>>
>>>>> Sorry about that.
>>>>>
>>>> I am as well, since I went to look at mem_cgroup_do_charge() and missed
>>>> that.
>>>
>>> I thought we are not doing atomic allocations in user pages accounting
>>> but I was obviously wrong because at least shmem uses atomic
>>> allocations for ages.
>>>
>>>> Do you have any other concerns specific to this patch ?
>>>
>>> I understood you changed also handle thingy. So the patch should be
>>> correct.
>>> Do you plan to send an updated version?
>>>

>> That depends more on you than on me! =)
>>
>> Do you still have any concerns regarding the u+k charging as it stands
>> now? That would be the last big concern I heard during this iteration.
>
> Well, I am still not 100% sure because I still see technical
> difficulties that are not addressed by the patchset (memcg-oom, memcg
> slab shrinking, possibly others). More importantly this is changing the
> current semantic of the limit so we should better be careful about it
> and check that we are not making the code tight to specific workloads
> without a way out.
>
> On the other hand I do not want to block the progress here without
> having _really_ good arguments against that couldn't be handled later
> (and it seems that some of my concerns are work in progress already).
>
> I have to admit I like several things about the patchset. Especially the
> way how it enables easy-to-setup (aka don't care about kmem details just
> make sure you can cap the thing) as well as "I know exactly what I want
> to do" usecases.
> It is also good nice that only users of the feature are affected by
> potential issues.
>
> So I think it is worth a broader attention which could produce other use
> cases which could show potential drawbacks from the u+k semantic but I
> would be still very careful about merging it to the Linus tree and only
> merge it after at least the memcg reclaim path is slab aware. Living in
> the -mm tree should help us with the testing converage.
>
> Does it sounds reasonable?
>
What I really want is to have it in an "official" tree so it starts
getting used and tested without me having to rebase at every single change.

If Andrew is okay merging this into -mm, it is fine for me.

Subject: Re: [PATCH v2 04/11] kmem accounting basic infrastructure
Posted by Michal Hocko on Thu, 16 Aug 2012 15:25:44 GMT
View Forum Message <> Reply to Message

On Wed 15-08-12 12:50:55, Ying Han wrote:
> On Tue, Aug 14, 2012 at 9:21 AM, Michal Hocko <mhocko@suse.cz> wrote:
> > On Thu 09-08-12 17:01:12, Glauber Costa wrote:
> >> This patch adds the basic infrastructure for the accounting of the slab
> >> caches. 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.
> >>
> >> The code is not enabled until a limit is set. This can be tested by the
> >> flag "kmem_accounted". This means that after the patch is applied, no
> >> behavioral changes exists for whoever is still using memcg to control
> >> their memory usage.
> >>
> >> 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.
> >
> > Well, it contributes to the user limit so it is not unlimited. It just
> > falls under a different limit and it tends to contribute less. This can
> > be quite confusing. I am still not sure whether we should mix the two
> > things together. If somebody wants to limit the kernel memory he has to
> > touch the other limit anyway. Do you have a strong reason to mix the
> > user and kernel counters?
>
> The reason to mix the two together is a compromise of the two use
> cases we've heard by far. In google, we only need one limit which
> limits u & k, and the reclaim kicks in when the total usage hits the
> limit.
>
> > My impression was that kernel allocation should simply fail while user
> > allocations might reclaim as well. Why should we reclaim just because of
> > the kernel allocation (which is unreclaimable from hard limit reclaim
> > point of view)?
>
> Some of kernel objects are reclaimable if we have per-memcg shrinker.

Agreed and I think we need that before this is merged as I state in
other email.

> > I also think that the whole thing would get much simpler if those two
> > are split. Anyway if this is really a must then this should be
> > documented here.
>
> What would be the use case you have in your end?

I do not have any specific unfortunately but I would like to prevent us
from closing other possible. I realize this sounds hand wavy and that is

why I do not want to block this work but I think we should give it some
time before this gets merged.

> --Ying
--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v2 06/11] memcg: kmem controller infrastructure
Posted by KAMEZAWA Hiroyuki on Fri, 17 Aug 2012 02:36:26 GMT
View Forum Message <> Reply to Message

(2012/08/13 17:28), Glauber Costa wrote:
>>>> + * Needs to be called after memcg_kmem_new_page, 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 @handle to the
>>>> + * corresponding page_cgroup.
>>>> + */
>>>> +static __always_inline void
>>>> +memcg_kmem_commit_page(struct page *page, struct mem_cgroup *handle, int order)
>>>> +{
>>>> + if (memcg_kmem_on)
>>>> + __memcg_kmem_commit_page(page, handle, order);
>>>> +}
>> Doesn't this 2 functions has no short-cuts ?
>
> Sorry kame, what exactly do you mean?
>
I meant avoinding function call. But please ignore, I missed following patches.

>> if (memcg_kmem_on && handle) ?
> I guess this can be done to avoid a function call.
>
>> Maybe free() needs to access page_cgroup...
>>
> Can you also be a bit more specific here?
>

Please ignore, I misunderstood the usage of free_accounted_pages().

>>>> +bool __memcg_kmem_new_page(gfp_t gfp, void *_handle, int order)
>>>> +{
>>>> + struct mem_cgroup *memcg;
>>>> + struct mem_cgroup **handle = (struct mem_cgroup **)_handle;
>>>> + bool ret = true;
>>>> + size_t size;

>>>> + struct task_struct *p;
>>>> +
>>>> + *handle = NULL;
>>>> + rcu_read_lock();
>>>> + p = rcu_dereference(current->mm->owner);
>>>> + memcg = mem_cgroup_from_task(p);
>>>> + if (!memcg_kmem_enabled(memcg))
>>>> + goto out;
>>>> +
>>>> + mem_cgroup_get(memcg);
>>>> +
>> This mem_cgroup_get() will be a potentioal performance problem.
>> Don't you have good idea to avoid accessing atomic counter here ?
>> I think some kind of percpu counter or a feature to disable "move task"
>> will be a help.
>
>
>
>
>>>> + pc = lookup_page_cgroup(page);
>>>> + lock_page_cgroup(pc);
>>>> + pc->mem_cgroup = memcg;
>>>> + SetPageCgroupUsed(pc);
>>>> + unlock_page_cgroup(pc);
>>>> +}
>>>> +
>>>> +void __memcg_kmem_free_page(struct page *page, int order)
>>>> +{
>>>> + struct mem_cgroup *memcg;
>>>> + size_t size;
>>>> + struct page_cgroup *pc;
>>>> +
>>>> + if (mem_cgroup_disabled())
>>>> + return;
>>>> +
>>>> + pc = lookup_page_cgroup(page);
>>>> + lock_page_cgroup(pc);
>>>> + memcg = pc->mem_cgroup;
>>>> + pc->mem_cgroup = NULL;
>
>> shouldn't this happen after checking "Used" bit ?
>> Ah, BTW, why do you need to clear pc->memcg ?
>
> As for clearing pc->memcg, I think I'm just being overzealous. I can't
> foresee any problems due to removing it.
>
> As for the Used bit, what difference does it make when we clear it?
>

I just want to see the same logic used in mem_cgroup_uncharge_common().
Hmm, at setting pc->mem_cgroup, the things happens in
  set pc->mem_cgroup
  set Used bit
order. If you clear pc->mem_cgroup
  unset Used bit
  clear pc->mem_cgroup
seems reasonable.

>>>> + if (!PageCgroupUsed(pc)) {
>>>> + unlock_page_cgroup(pc);
>>>> + return;
>>>> + }
>>>> + ClearPageCgroupUsed(pc);
>>>> + unlock_page_cgroup(pc);
>>>> +
>>>> + /*
>>>> + * Checking if kmem accounted is enabled won't work for uncharge, since
>>>> + * it is possible that the user enabled kmem tracking, allocated, and
>>>> + * then disabled it again.
>>>> + *
>>>> + * We trust if there is a memcg associated with the page, it is a valid
>>>> + * allocation
>>>> + */
>>>> + if (!memcg)
>>>> + return;
>>>> +
>>>> + WARN_ON(mem_cgroup_is_root(memcg));
>>>> + size = (1 << order) << PAGE_SHIFT;
>>>> + memcg_uncharge_kmem(memcg, size);
>>>> + mem_cgroup_put(memcg);
>> Why do we need ref-counting here ? kmem res_counter cannot work as
>> reference ?
> This is of course the pair of the mem_cgroup_get() you commented on
> earlier. If we need one, we need the other. If we don't need one, we
> don't need the other =)
>
> The guarantee we're trying to give here is that the memcg structure will
> stay around while there are dangling charges to kmem, that we decided
> not to move (remember: moving it for the stack is simple, for the slab
> is very complicated and ill-defined, and I believe it is better to treat
> all kmem equally here)
>
> So maybe we can be clever here, and avoid reference counting at all
> times. We call mem_cgroup_get() when the first charge occurs, and then
> go for mem_cgroup_put() when our count reaches 0.

>
> What do you think about that?
>

I think that should work. I don't want to add not-optimized atomic counter ops
in this very hot path.

>
>>>> +#ifdef CONFIG_MEMCG_KMEM
>>>> +int memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t gfp, s64 delta)
>>>> +{
>> What does 'delta' means ?
>>
> I can change it to something like nr_bytes, more informative.
>
>>>> + struct res_counter *fail_res;
>>>> + struct mem_cgroup *_memcg;
>>>> + int ret;
>>>> + bool may_oom;
>>>> + bool nofail = false;
>>>> +
>>>> + may_oom = (gfp & __GFP_WAIT) && (gfp & __GFP_FS) &&
>>>> +   !(gfp & __GFP_NORETRY);
>>>> +
>>>> + ret = 0;
>>>> +
>>>> + if (!memcg)
>>>> + return ret;
>>>> +
>>>> + _memcg = memcg;
>>>> + ret = __mem_cgroup_try_charge(NULL, gfp, delta / PAGE_SIZE,
>>>> +   &_memcg, may_oom);
>>>> +
>>>> + if (ret == -EINTR) {
>>>> + nofail = true;
>>>> + /*
>>>> +   * __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
>>>> +   */
>>>> + res_counter_charge_nofail(&memcg->res, delta, &fail_res);
>>>> + if (do_swap_account)
>>>> +   res_counter_charge_nofail(&memcg->memsw, delta,

>>>> + &fail_res);
>>>> + ret = 0;
>> Hm, you returns 0 and this charge may never be uncharged....right ?
>>
>
> Can't see why. By returning 0 we inform our caller that the allocation
> succeeded. It is up to him to undo it later through a call to uncharge.
>
Hmm, okay. You trust callers.

Thanks,
-Kame

Subject: Re: [PATCH v2 04/11] kmem accounting basic infrastructure
Posted by KAMEZAWA Hiroyuki on Fri, 17 Aug 2012 02:38:34 GMT
View Forum Message <> Reply to Message

(2012/08/13 17:36), Glauber Costa wrote:
> On 08/10/2012 09:02 PM, Kamezawa Hiroyuki wrote:
>> (2012/08/09 22:01), Glauber Costa wrote:
>>> This patch adds the basic infrastructure for the accounting of the slab
>>> caches. 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.
>>>
>>> The code is not enabled until a limit is set. This can be tested by the
>>> flag "kmem_accounted". This means that after the patch is applied, no
>>> behavioral changes exists for whoever is still using memcg to control
>>> their memory usage.
>>>
>>> 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)
>>>
>>> Signed-off-by: Glauber Costa <glommer@parallels.com>

>>> CC: Michal Hocko <mhocko@suse.cz>
>>> CC: Johannes Weiner <hannes@cmpxchg.org>
>>> Reviewed-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
>>
>> Could you add a patch for documentation of this new interface and a text
>> explaining the behavior of "kmem_accounting" ?
>>
>> Hm, my concern is the difference of behavior between user page accounting and
>> kmem accounting...but this is how tcp-accounting is working.
>>
>> Once you add Documentation, it's okay to add my Ack.
>>
> I plan to add documentation in a separate patch. Due to that, can I add
> your ack to this patch here?
>
> Also, I find that the description text in patch0 grew to be quite
> informative and complete. I plan to add that to the documentation
> if that is ok with you
>

Ack to this patch.

-Kame

Subject: Re: [PATCH v2 04/11] kmem accounting basic infrastructure
Posted by Ying Han on Fri, 17 Aug 2012 05:58:21 GMT
View Forum Message <> Reply to Message

On Thu, Aug 16, 2012 at 8:25 AM, Michal Hocko <mhocko@suse.cz> wrote:
> On Wed 15-08-12 12:50:55, Ying Han wrote:
>> On Tue, Aug 14, 2012 at 9:21 AM, Michal Hocko <mhocko@suse.cz> wrote:
>> > On Thu 09-08-12 17:01:12, Glauber Costa wrote:
>> >> This patch adds the basic infrastructure for the accounting of the slab
>> >> caches. 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.
>> >>
>> >> The code is not enabled until a limit is set. This can be tested by the
>> >> flag "kmem_accounted". This means that after the patch is applied, no
>> >> behavioral changes exists for whoever is still using memcg to control
>> >> their memory usage.

>> >>
>> >> 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.
>> >
>> > Well, it contributes to the user limit so it is not unlimited. It just
>> > falls under a different limit and it tends to contribute less. This can
>> > be quite confusing. I am still not sure whether we should mix the two
>> > things together. If somebody wants to limit the kernel memory he has to
>> > touch the other limit anyway. Do you have a strong reason to mix the
>> > user and kernel counters?
>>
>> The reason to mix the two together is a compromise of the two use
>> cases we've heard by far. In google, we only need one limit which
>> limits u & k, and the reclaim kicks in when the total usage hits the
>> limit.
>>
>> > My impression was that kernel allocation should simply fail while user
>> > allocations might reclaim as well. Why should we reclaim just because of
>> > the kernel allocation (which is unreclaimable from hard limit reclaim
>> > point of view)?
>>
>> Some of kernel objects are reclaimable if we have per-memcg shrinker.
>
> Agreed and I think we need that before this is merged as I state in
> other email.
>
>> > I also think that the whole thing would get much simpler if those two
>> > are split. Anyway if this is really a must then this should be
>> > documented here.
>>
>> What would be the use case you have in your end?
>
> I do not have any specific unfortunately but I would like to prevent us
> from closing other possible. I realize this sounds hand wavy and that is
> why I do not want to block this work but I think we should give it some
> time before this gets merged.

Agreed that we don't want to rush merge anything.

On the other hand, I was trying to understand your concern of the k &
u+k counter. After reading your previous replies,
I think I understand your concern of missing the target shrinker. I
posted the patch and please take a look :)

Meanwhile, can you help to clarify other concerns in your mind on

having the two counters? Please ignore me if you answered
the question somewhere and just give me the pointer.

--Ying

>
>> --Ying
> --
> Michal Hocko
> SUSE Labs

Subject: Re: [PATCH v2 08/11] memcg: disable kmem code when not in use.
Posted by Glauber Costa on Fri, 17 Aug 2012 07:01:06 GMT
View Forum Message <> Reply to Message

On 08/17/2012 11:02 AM, Michal Hocko wrote:
> On Thu 09-08-12 17:01:16, Glauber Costa wrote:
>> We can use jump labels to patch the code in or out when not used.
>>
>> Because the assignment: memcg->kmem_accounted = true is done after the
>> jump labels increment, we guarantee that the root memcg will always be
>> selected until all call sites are patched (see memcg_kmem_enabled).
>
> Not that it would be really important because kmem_accounted goes away
> in a subsequent patch but I think the wording is a bit misleading here.
> First of all there is no guanratee that kmem_accounted=true is seen
> before atomic_inc(&key->enabled) because there is no memory barrier and
> the lock serves just a leave barrier. But I do not think this is
> important at all because key->enabled is what matters here. Even if
> memcg_kmem_enabled is true we do not consider it if the key is disabled,
> right?
>

Right.

Subject: Re: [PATCH v2 08/11] memcg: disable kmem code when not in use.
Posted by Michal Hocko on Fri, 17 Aug 2012 07:02:41 GMT
View Forum Message <> Reply to Message

On Thu 09-08-12 17:01:16, Glauber Costa wrote:
> We can use jump labels to patch the code in or out when not used.
>
> Because the assignment: memcg->kmem_accounted = true is done after the
> jump labels increment, we guarantee that the root memcg will always be
> selected until all call sites are patched (see memcg_kmem_enabled).

Not that it would be really important because kmem_accounted goes away
in a subsequent patch but I think the wording is a bit misleading here.
First of all there is no guanratee that kmem_accounted=true is seen
before atomic_inc(&key->enabled) because there is no memory barrier and
the lock serves just a leave barrier. But I do not think this is
important at all because key->enabled is what matters here. Even if
memcg_kmem_enabled is true we do not consider it if the key is disabled,
right?

> This guarantees that no mischarges are applied.
>
> Jump label decrement happens when the last reference count from the
> memcg dies. This will only happen when the caches are all dead.
>
> Signed-off-by: Glauber Costa <glommer@parallels.com>
> 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: Johannes Weiner <hannes@cmpxchg.org>
> CC: Suleiman Souhlal <suleiman@google.com>

Anyway the code looks correct.
Reviewed-by: Michal Hocko <mhocko@suse.cz>

> ---
> include/linux/memcontrol.h | 5 ++++-
> mm/memcontrol.c | 50 ++----------
> 2 files changed, 44 insertions(+), 11 deletions(-)
>
> diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
> index 75b247e..f39d933 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;
> @@ -401,7 +402,9 @@ struct sock;
> void sock_update_memcg(struct sock *sk);
> void sock_release_memcg(struct sock *sk);
>
> -#define memcg_kmem_on 1

> +extern struct static_key memcg_kmem_enabled_key;
> +#define memcg_kmem_on static_key_false(&memcg_kmem_enabled_key)
> +
> bool __memcg_kmem_new_page(gfp_t gfp, void *handle, int order);
> void __memcg_kmem_commit_page(struct page *page, void *handle, int order);
> void __memcg_kmem_free_page(struct page *page, int order);
> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
> index e9824c1..3216292 100644
> --- a/mm/memcontrol.c
> +++ b/mm/memcontrol.c
> @@ -437,6 +437,10 @@ struct mem_cgroup *mem_cgroup_from_css(struct
cgroup_subsys_state *s)
> #include <net/sock.h>
> #include <net/ip.h>
>
> +struct static_key memcg_kmem_enabled_key;
> +/* so modules can inline the checks */
> +EXPORT_SYMBOL(memcg_kmem_enabled_key);
> +
> static bool mem_cgroup_is_root(struct mem_cgroup *memcg);
> static int memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t gfp, s64 delta);
> static void memcg_uncharge_kmem(struct mem_cgroup *memcg, s64 delta);
> @@ -607,6 +611,16 @@ void __memcg_kmem_free_page(struct page *page, int order)
> mem_cgroup_put(memcg);
> }
> EXPORT_SYMBOL(__memcg_kmem_free_page);
> +
> +static void disarm_kmem_keys(struct mem_cgroup *memcg)
> +{
> + if (memcg->kmem_accounted)
> + static_key_slow_dec(&memcg_kmem_enabled_key);
> +}
> +#else
> +static void disarm_kmem_keys(struct mem_cgroup *memcg)
> +{
> +}
> #endif /* CONFIG_MEMCG_KMEM */
>
> #if defined(CONFIG_INET) && defined(CONFIG_MEMCG_KMEM)
> @@ -622,6 +636,12 @@ static void disarm_sock_keys(struct mem_cgroup *memcg)
> }
> #endif
>
> +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 *
> @@ -4147,6 +4167,24 @@ 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 void memcg_update_kmem_limit(struct mem_cgroup *memcg, u64 val)
> +{
> +#ifdef CONFIG_MEMCG_KMEM
> + /*
> +  * Once enabled, can't be disabled. We could in theory disable it if we
> +  * haven't yet created any caches, or if we can shrink them all to
> +  * death. But it is not worth the trouble.
> +  */
> + mutex_lock(&set_limit_mutex);
> + if (!memcg->kmem_accounted && val != RESOURCE_MAX) {
> +  static_key_slow_inc(&memcg_kmem_enabled_key);
> +  memcg->kmem_accounted = true;
> + }
> + mutex_unlock(&set_limit_mutex);
> +#endif
> +}
> +
>  /*
>   * The user of this function is...
>   * RES_LIMIT.
> @@ -4184,15 +4222,7 @@ static int mem_cgroup_write(struct cgroup *cont, struct cftype *cft,
>     ret = res_counter_set_limit(&memcg->kmem, val);
>     if (ret)
>      break;
> -   /*
> -    * Once enabled, can't be disabled. We could in theory
> -    * disable it if we haven't yet created any caches, or
> -    * if we can shrink them all to death.
> -    *
> -    * But it is not worth the trouble
> -    */
> -   if (!memcg->kmem_accounted && val != RESOURCE_MAX)
> -    memcg->kmem_accounted = true;
> +   memcg_update_kmem_limit(memcg, val);
>    } else
>     return -EINVAL;
>    break;
> @@ -5054,7 +5084,7 @@ 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
> --
> 1.7.11.2
>
> --
> To unsubscribe from this list: send the line "unsubscribe cgroups" in
> the body of a message to majordomo@vger.kernel.org
> More majordomo info at http://vger.kernel.org/majordomo-info.html

--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v2 06/11] memcg: kmem controller infrastructure
Posted by Glauber Costa on Fri, 17 Aug 2012 07:04:02 GMT
View Forum Message <> Reply to Message

On 08/17/2012 06:36 AM, Kamezawa Hiroyuki wrote:
> I just want to see the same logic used in mem_cgroup_uncharge_common().
> Hmm, at setting pc->mem_cgroup, the things happens in
> set pc->mem_cgroup
> set Used bit
> order. If you clear pc->mem_cgroup
> unset Used bit
> clear pc->mem_cgroup
> seems reasonable.

Makes sense. I'll make sure we're consistent here.

Subject: Re: [PATCH v2 08/11] memcg: disable kmem code when not in use.
Posted by Michal Hocko on Fri, 17 Aug 2012 08:04:04 GMT
View Forum Message <> Reply to Message

On Fri 17-08-12 11:01:06, Glauber Costa wrote:
> On 08/17/2012 11:02 AM, Michal Hocko wrote:
> > On Thu 09-08-12 17:01:16, Glauber Costa wrote:
> >> We can use jump labels to patch the code in or out when not used.
> >>

> >> Because the assignment: memcg->kmem_accounted = true is done after the
> >> jump labels increment, we guarantee that the root memcg will always be
> >> selected until all call sites are patched (see memcg_kmem_enabled).
> >
> > Not that it would be really important because kmem_accounted goes away

And just found out it doesn't go away completely, it just transforms
from bool to unsigned log (with flags). The rest still holds...

> > in a subsequent patch but I think the wording is a bit misleading here.
> > First of all there is no guanratee that kmem_accounted=true is seen
> > before atomic_inc(&key->enabled) because there is no memory barrier and
> > the lock serves just a leave barrier. But I do not think this is
> > important at all because key->enabled is what matters here. Even if
> > memcg_kmem_enabled is true we do not consider it if the key is disabled,
> > right?
> >
>
> Right.

--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v2 09/11] memcg: propagate kmem limiting information to children
Posted by Michal Hocko on Fri, 17 Aug 2012 09:00:06 GMT
View Forum Message <> Reply to Message

On Thu 09-08-12 17:01:17, Glauber Costa wrote:
> The current memcg slab cache management fails to present satisfatory
> hierarchical behavior in the following scenario:
>
> -> /cgroups/memory/A/B/C
>
> * kmem limit set at A,
> * A and B have no tasks,
> * span a new task in in C.
>
> Because kmem_accounted is a boolean that was not set for C, no
> accounting would be done. This is, however, not what we expect.
>
> The basic idea, is that when a cgroup is limited, we walk the tree
> upwards

Isn't it rather downwards? We start at A and then mark all children so
we go down the tree. Moreover the walk is not atomic wrt. parallel

charges nor to a new child creation. First one seems to be acceptable
as the charges go to the root. The second one requires cgroup_lock.

It also seems that you are missing memcg_kmem_account_parent in
mem_cgroup_create (use_hierarchy path) if memcg_kmem_is_accounted(parent).

Some further "wording" comments below. Other than that the patch looks
correct.

> (something Kame and I already thought about doing for other
> purposes), and make sure that we store the information about the parent
> being limited in kmem_accounted (that is turned into a bitmap: two
> booleans would not be space efficient).

Two booleans even don't serve the purpose because you want to test this
atomically, right?

> The code for that is taken from sched/core.c. My reasons for not
> putting it into a common place is to dodge the type issues that would
> arise from a common implementation between memcg and the scheduler -
> but I think that it should ultimately happen, so if you want me to do
> it now, let me know.

Is this really relevant for the patch?

> We do the reverse operation when a formerly limited cgroup becomes
> unlimited.
>
> Signed-off-by: Glauber Costa <glommer@parallels.com>
> CC: Christoph Lameter <cl@linux.com>
> CC: Pekka Enberg <penberg@cs.helsinki.fi>
> CC: Michal Hocko <mhocko@suse.cz>
> CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
> CC: Johannes Weiner <hannes@cmpxchg.org>
> CC: Suleiman Souhlal <suleiman@google.com>
> ---
> mm/memcontrol.c | 88 ++------
> 1 file changed, 79 insertions(+), 9 deletions(-)
>
> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
> index 3216292..3d30b79 100644
> --- a/mm/memcontrol.c
> +++ b/mm/memcontrol.c
> @@ -295,7 +295,8 @@ struct mem_cgroup {
>   * Should the accounting and control be hierarchical, per subtree?
>   */
>   bool use_hierarchy;
> - bool kmem_accounted;

> +
> + unsigned long kmem_accounted; /* See KMEM_ACCOUNTED_*, below */
>
> bool oom_lock;
> atomic_t under_oom;
> @@ -348,6 +349,38 @@ struct mem_cgroup {
> #endif
> };
>
> +enum {
> + KMEM_ACCOUNTED_THIS, /* accounted by this cgroup itself */
> + KMEM_ACCOUNTED_PARENT, /* accounted by any of its parents. */

How it can be accounted by its parent, the charge doesn't go downwards.
Shouldn't it rather be /* a parent is accounted */

> +};
> +
> +#ifdef CONFIG_MEMCG_KMEM
> +static bool memcg_kmem_account(struct mem_cgroup *memcg)

memcg_kmem_set_account? It matches _clear_ counterpart and it makes
obvious that the value is changed actually.

[...]
> +static bool memcg_kmem_is_accounted(struct mem_cgroup *memcg)
> +{
> + return test_bit(KMEM_ACCOUNTED_THIS, &memcg->kmem_accounted);
> +}
> +
> +static void memcg_kmem_account_parent(struct mem_cgroup *memcg)

same here _set_parent

[...]
> @@ -614,7 +647,7 @@ EXPORT_SYMBOL(__memcg_kmem_free_page);
>
> static void disarm_kmem_keys(struct mem_cgroup *memcg)
> {
> - if (memcg->kmem_accounted)
> + if (test_bit(KMEM_ACCOUNTED_THIS, &memcg->kmem_accounted))

memcg_kmem_is_accounted. I do not see any reason to open code this.

> static_key_slow_dec(&memcg_kmem_enabled_key);
> }
> #else
> @@ -4171,17 +4204,54 @@ static ssize_t mem_cgroup_read(struct cgroup *cont, struct cftype

*cft,
> static void memcg_update_kmem_limit(struct mem_cgroup *memcg, u64 val)
> {
> #ifdef CONFIG_MEMCG_KMEM
> - /*
> - * Once enabled, can't be disabled. We could in theory disable it if we
> - * haven't yet created any caches, or if we can shrink them all to
> - * death. But it is not worth the trouble.
> - */
> + struct mem_cgroup *iter;
> +
>   mutex_lock(&set_limit_mutex);
> - if (!memcg->kmem_accounted && val != RESOURCE_MAX) {
> + if ((val != RESOURCE_MAX) && memcg_kmem_account(memcg)) {
> +
> + /*
> +   * Once enabled, can't be disabled. We could in theory disable
> +   * it if we haven't yet created any caches, or if we can shrink
> +   * them all to death. But it is not worth the trouble
> +   */
>   static_key_slow_inc(&memcg_kmem_enabled_key);
> - memcg->kmem_accounted = true;
> +
> + if (!memcg->use_hierarchy)
> +   goto out;
> +
> + for_each_mem_cgroup_tree(iter, memcg) {

for_each_mem_cgroup_tree does respect use_hierarchy so the above
shortcut is not necessary. Dunno but IMHO we should get rid of explicit
tests as much as possible. This doesn't look like a hot path anyway.

> +   if (iter == memcg)
> +   continue;
> +   memcg_kmem_account_parent(iter);
> + }
> + } else if ((val == RESOURCE_MAX) && memcg_kmem_clear_account(memcg)) {

Above you said "Once enabled, can't be disabled." and now you can
disable it? Say you are a leaf group with non accounted parents. This
will clear the flag and so no further accounting is done. Shouldn't
unlimited mean that we will never reach the limit? Or am I missing
something?

> +
> + if (!memcg->use_hierarchy)
> +   goto out;
> +

```
> +  for_each_mem_cgroup_tree(iter, memcg) {
> +   struct mem_cgroup *parent;
> +
> +   if (iter == memcg)
> +    continue;
> +   /*
> +    * We should only have our parent bit cleared if none
> +    * of our parents are accounted. The transversal order
> +    * of our iter function forces us to always look at the
> +    * parents.
> +    */
> +   parent = parent_mem_cgroup(iter);
> +   for (; parent != memcg; parent = parent_mem_cgroup(iter))
> +    if (memcg_kmem_is_accounted(parent))
> +     goto noclear;
> +   memcg_kmem_clear_account_parent(iter);
```

Brain hurts...
Yes we are iterating in the creation ordering so we cannot rely on the
first encountered accounted memcg
A(a) - B - D
    - C (a) - E

```
> +noclear:
> +   continue;
> +  }
>   }
> +out:
>   mutex_unlock(&set_limit_mutex);
> +
>  #endif
>  }
>
> --
> 1.7.11.2
>
> --
> To unsubscribe from this list: send the line "unsubscribe cgroups" in
> the body of a message to majordomo@vger.kernel.org
> More majordomo info at  http://vger.kernel.org/majordomo-info.html
```

--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v2 09/11] memcg: propagate kmem limiting information to children
Posted by Glauber Costa on Fri, 17 Aug 2012 09:15:47 GMT
View Forum Message <> Reply to Message

On 08/17/2012 01:00 PM, Michal Hocko wrote:
> On Thu 09-08-12 17:01:17, Glauber Costa wrote:
>> The current memcg slab cache management fails to present satisfatory
>> hierarchical behavior in the following scenario:
>>
>> -> /cgroups/memory/A/B/C
>>
>> * kmem limit set at A,
>> * A and B have no tasks,
>> * span a new task in in C.
>>
>> Because kmem_accounted is a boolean that was not set for C, no
>> accounting would be done. This is, however, not what we expect.
>>
>> The basic idea, is that when a cgroup is limited, we walk the tree
>> upwards
>
> Isn't it rather downwards? We start at A and then mark all children so
> we go down the tree. Moreover the walk is not atomic wrt. parallel
> charges nor to a new child creation. First one seems to be acceptable
> as the charges go to the root. The second one requires cgroup_lock.
>

Yes, it is downwards. I've already noticed that yesterday and updated
in my tree.

As for the lock, can't we take set_limit lock in cgroup creation just
around the place that updates that field in the child? It is a lot more
fine grained - everything except the dead bkl is - and what we're
actually protecting is the limit.

If you prefer, I can use cgroup lock just fine. But then I won't sleep
at night and probably pee my pants, which is something I don't do for at
least two decades now.

> It also seems that you are missing memcg_kmem_account_parent in
> mem_cgroup_create (use_hierarchy path) if memcg_kmem_is_accounted(parent).
>

You mean when we create a cgroup ontop of an already limited parent?
Humm, you are very right.

> Some further "wording" comments below. Other than that the patch looks
> correct.

>
>> (something Kame and I already thought about doing for other
>> purposes), and make sure that we store the information about the parent
>> being limited in kmem_accounted (that is turned into a bitmap: two
>> booleans would not be space efficient).
>
> Two booleans even don't serve the purpose because you want to test this
> atomically, right?
>

Well, yes, we have that extra problem as well.
>> The code for that is taken from sched/core.c. My reasons for not
>> putting it into a common place is to dodge the type issues that would
>> arise from a common implementation between memcg and the scheduler -
>> but I think that it should ultimately happen, so if you want me to do
>> it now, let me know.
>
> Is this really relevant for the patch?
>

Not at all. Besides not being relevant, it is also not true, since I now
use the memcg iterator. I would prefer the tree walk instead of having
to cope with the order imposed by the memcg iterator, but we add
less code this way...

Again, already modified that in my yesterday's update.

>> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
>> index 3216292..3d30b79 100644
>> --- a/mm/memcontrol.c
>> +++ b/mm/memcontrol.c
>> @@ -295,7 +295,8 @@ struct mem_cgroup {
>>   * Should the accounting and control be hierarchical, per subtree?
>>   */
>>   bool use_hierarchy;
>> - bool kmem_accounted;
>> +
>> + unsigned long kmem_accounted; /* See KMEM_ACCOUNTED_*, below */
>>
>>   bool oom_lock;
>>   atomic_t under_oom;
>> @@ -348,6 +349,38 @@ struct mem_cgroup {
>> #endif
>> };
>>
>> +enum {
>> + KMEM_ACCOUNTED_THIS, /* accounted by this cgroup itself */
>> + KMEM_ACCOUNTED_PARENT, /* accounted by any of its parents. */

>
> How it can be accounted by its parent, the charge doesn't go downwards.
> Shouldn't it rather be /* a parent is accounted */
>
indeed.

>> +};
>> +
>> +#ifdef CONFIG_MEMCG_KMEM
>> +static bool memcg_kmem_account(struct mem_cgroup *memcg)
>
> memcg_kmem_set_account? It matches _clear_ counterpart and it makes
> obvious that the value is changed actually.
>

Ok.

> [...]
>> +static bool memcg_kmem_is_accounted(struct mem_cgroup *memcg)
>> +{
>> + return test_bit(KMEM_ACCOUNTED_THIS, &memcg->kmem_accounted);
>> +}
>> +
>> +static void memcg_kmem_account_parent(struct mem_cgroup *memcg)
>
> same here _set_parent
>

Ok, agreed.

> [...]
>> @@ -614,7 +647,7 @@ EXPORT_SYMBOL(__memcg_kmem_free_page);
>>
>> static void disarm_kmem_keys(struct mem_cgroup *memcg)
>> {
>> - if (memcg->kmem_accounted)
>> + if (test_bit(KMEM_ACCOUNTED_THIS, &memcg->kmem_accounted))
>
> memcg_kmem_is_accounted. I do not see any reason to open code this.
>

ok.

>> #ifdef CONFIG_MEMCG_KMEM
>> - /*
>> - * Once enabled, can't be disabled. We could in theory disable it if we
>> - * haven't yet created any caches, or if we can shrink them all to
>> - * death. But it is not worth the trouble.

>> - */
>> + struct mem_cgroup *iter;
>> +
>>   mutex_lock(&set_limit_mutex);
>> - if (!memcg->kmem_accounted && val != RESOURCE_MAX) {
>> + if ((val != RESOURCE_MAX) && memcg_kmem_account(memcg)) {
>> +
>> + /*
>> +   * Once enabled, can't be disabled. We could in theory disable
>> +   * it if we haven't yet created any caches, or if we can shrink
>> +   * them all to death. But it is not worth the trouble
>> +   */
>>   static_key_slow_inc(&memcg_kmem_enabled_key);
>> - memcg->kmem_accounted = true;
>> +
>> + if (!memcg->use_hierarchy)
>> +   goto out;
>> +
>> + for_each_mem_cgroup_tree(iter, memcg) {
>
> for_each_mem_cgroup_tree does respect use_hierarchy so the above
> shortcut is not necessary. Dunno but IMHO we should get rid of explicit
> tests as much as possible. This doesn't look like a hot path anyway.
>

I can't remember any reason for doing so other than gaining some time.
I will remove it.

>> +   if (iter == memcg)
>> +   continue;
>> +   memcg_kmem_account_parent(iter);
>> + }
>> + } else if ((val == RESOURCE_MAX) && memcg_kmem_clear_account(memcg)) {
>
> Above you said "Once enabled, can't be disabled." and now you can
> disable it? Say you are a leaf group with non accounted parents. This
> will clear the flag and so no further accounting is done. Shouldn't
> unlimited mean that we will never reach the limit? Or am I missing
> something?
>

You are missing something, and maybe I should be more clear about that.
The static branches can't be disabled (it is only safe to disable them
from disarm_static_branches(), when all references are gone). Note that
when unlimited, we flip bits, do a transversal, but there is no mention
to the static branch.

The limiting can come and go at will.

```
>> +
>> +  if (!memcg->use_hierarchy)
>> +   goto out;
>> +
>> +  for_each_mem_cgroup_tree(iter, memcg) {
>> +   struct mem_cgroup *parent;
>> +
>> +   if (iter == memcg)
>> +    continue;
>> +   /*
>> +    * We should only have our parent bit cleared if none
>> +    * of our parents are accounted. The transversal order
>> +    * of our iter function forces us to always look at the
>> +    * parents.
>> +    */
>> +   parent = parent_mem_cgroup(iter);
>> +   for (; parent != memcg; parent = parent_mem_cgroup(iter))
>> +    if (memcg_kmem_is_accounted(parent))
>> +     goto noclear;
>> +   memcg_kmem_clear_account_parent(iter);
>
> Brain hurts...
> Yes we are iterating in the creation ordering so we cannot rely on the
> first encountered accounted memcg
> A(a) - B - D
>      - C (a) - E
>
>
```

That's why I said I preferred the iterator the scheduler uses. The actual transverse code was much simpler, because it will stop at an unlimited parent. But this is the only drawback I see in the memcg iterator, so I decided that just documenting this "interesting" piece of code well would do...

Subject: Re: [PATCH v2 09/11] memcg: propagate kmem limiting information to children
Posted by Michal Hocko on Fri, 17 Aug 2012 09:35:04 GMT
View Forum Message <> Reply to Message

On Fri 17-08-12 13:15:47, Glauber Costa wrote:
> On 08/17/2012 01:00 PM, Michal Hocko wrote:
> > On Thu 09-08-12 17:01:17, Glauber Costa wrote:
> >> The current memcg slab cache management fails to present satisfatory
> >> hierarchical behavior in the following scenario:
> >>

> >> -> /cgroups/memory/A/B/C
> >>
> >> * kmem limit set at A,
> >> * A and B have no tasks,
> >> * span a new task in in C.
> >>
> >> Because kmem_accounted is a boolean that was not set for C, no
> >> accounting would be done. This is, however, not what we expect.
> >>
> >> The basic idea, is that when a cgroup is limited, we walk the tree
> >> upwards
> >
> > Isn't it rather downwards? We start at A and then mark all children so
> > we go down the tree. Moreover the walk is not atomic wrt. parallel
> > charges nor to a new child creation. First one seems to be acceptable
> > as the charges go to the root. The second one requires cgroup_lock.
> >
>
> Yes, it is downwards. I've already noticed that yesterday and updated
> in my tree.
>
> As for the lock, can't we take set_limit lock in cgroup creation just
> around the place that updates that field in the child? It is a lot more
> fine grained - everything except the dead bkl is - and what we're
> actually protecting is the limit.

That should work as well. It is less obvious because we are not
considering the parent limit (maybe we should rename the lock but that
is just a detail).

> If you prefer, I can use cgroup lock just fine. But then I won't sleep
> at night and probably pee my pants, which is something I don't do for at
> least two decades now.

Heh, please no, I would feel terrible then

> > It also seems that you are missing memcg_kmem_account_parent in
> > mem_cgroup_create (use_hierarchy path) if memcg_kmem_is_accounted(parent).
> >
>
> You mean when we create a cgroup ontop of an already limited parent?

I would prefer bellow but yes
A (a) - B (a, pa)
      - C (new)

> Humm, you are very right.
>

> > Some further "wording" comments below. Other than that the patch looks
> > correct.
> >
> >> (something Kame and I already thought about doing for other
> >> purposes), and make sure that we store the information about the parent
> >> being limited in kmem_accounted (that is turned into a bitmap: two
> >> booleans would not be space efficient).
> >
> > Two booleans even don't serve the purpose because you want to test this
> > atomically, right?
> >
>
> Well, yes, we have that extra problem as well.
> >> The code for that is taken from sched/core.c. My reasons for not
> >> putting it into a common place is to dodge the type issues that would
> >> arise from a common implementation between memcg and the scheduler -
> >> but I think that it should ultimately happen, so if you want me to do
> >> it now, let me know.
> >
> > Is this really relevant for the patch?
> >
>
> Not at all. Besides not being relevant, it is also not true, since I now
> use the memcg iterator. I would prefer the tree walk instead of having
> to cope with the order imposed by the memcg iterator, but we add
> less code this way...
>
> Again, already modified that in my yesterday's update.

OK

> >> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
> >> index 3216292..3d30b79 100644
> >> --- a/mm/memcontrol.c
> >> +++ b/mm/memcontrol.c
> >> @@ -295,7 +295,8 @@ struct mem_cgroup {
> >>   * Should the accounting and control be hierarchical, per subtree?
> >>   */
> >>   bool use_hierarchy;
> >> - bool kmem_accounted;
> >> +
> >> + unsigned long kmem_accounted; /* See KMEM_ACCOUNTED_*, below */
> >>
> >>   bool oom_lock;
> >>   atomic_t under_oom;
> >> @@ -348,6 +349,38 @@ struct mem_cgroup {
> >> #endif
> >> };

> >>
> >> +enum {
> >> + KMEM_ACCOUNTED_THIS, /* accounted by this cgroup itself */
> >> + KMEM_ACCOUNTED_PARENT, /* accounted by any of its parents. */
> >
> > How it can be accounted by its parent, the charge doesn't go downwards.
> > Shouldn't it rather be /* a parent is accounted */
> >
> indeed.
>
> >> +};
> >> +
> >> +#ifdef CONFIG_MEMCG_KMEM
> >> +static bool memcg_kmem_account(struct mem_cgroup *memcg)
> >
> > memcg_kmem_set_account? It matches _clear_ counterpart and it makes
> > obvious that the value is changed actually.
> >
> >
> Ok.
>
> > [...]
> >> +static bool memcg_kmem_is_accounted(struct mem_cgroup *memcg)
> >> +{
> >> + return test_bit(KMEM_ACCOUNTED_THIS, &memcg->kmem_accounted);
> >> +}
> >> +
> >> +static void memcg_kmem_account_parent(struct mem_cgroup *memcg)
> >
> > same here _set_parent
> >
> >
> Ok, agreed.

Thanks

>
> > [...]
> >> @@ -614,7 +647,7 @@ EXPORT_SYMBOL(__memcg_kmem_free_page);
> >>
> >> static void disarm_kmem_keys(struct mem_cgroup *memcg)
> >> {
> >> - if (memcg->kmem_accounted)
> >> + if (test_bit(KMEM_ACCOUNTED_THIS, &memcg->kmem_accounted))
> >
> > memcg_kmem_is_accounted. I do not see any reason to open code this.
> >
> >
>

> ok.
>
> >> #ifdef CONFIG_MEMCG_KMEM
> >> - /*
> >> - * Once enabled, can't be disabled. We could in theory disable it if we
> >> - * haven't yet created any caches, or if we can shrink them all to
> >> - * death. But it is not worth the trouble.
> >> - */
> >> + struct mem_cgroup *iter;
> >> +
> >>   mutex_lock(&set_limit_mutex);
> >> - if (!memcg->kmem_accounted && val != RESOURCE_MAX) {
> >> + if ((val != RESOURCE_MAX) && memcg_kmem_account(memcg)) {
> >> +
> >> + /*
> >> +   * Once enabled, can't be disabled. We could in theory disable
> >> +   * it if we haven't yet created any caches, or if we can shrink
> >> +   * them all to death. But it is not worth the trouble
> >> +   */
> >>   static_key_slow_inc(&memcg_kmem_enabled_key);
> >> - memcg->kmem_accounted = true;
> >> +
> >> + if (!memcg->use_hierarchy)
> >> +   goto out;
> >> +
> >> + for_each_mem_cgroup_tree(iter, memcg) {
> >
> > for_each_mem_cgroup_tree does respect use_hierarchy so the above
> > shortcut is not necessary. Dunno but IMHO we should get rid of explicit
> > tests as much as possible. This doesn't look like a hot path anyway.
> >
>
> I can't remember any reason for doing so other than gaining some time.
> I will remove it.

Well it involves a bit more code because you would basically do expand
to a loop which does one iteration (continue) and terminates also take
and drop the reference on the group. That all seems unnecessary but as I
said this is not a hot path and we better get rid of direct checks.
I am not insisting on this so use your good taste...

>
> >> +   if (iter == memcg)
> >> +   continue;
> >> +   memcg_kmem_account_parent(iter);
> >> + }
> >> + } else if ((val == RESOURCE_MAX) && memcg_kmem_clear_account(memcg)) {
> >

> > Above you said "Once enabled, can't be disabled." and now you can
> > disable it? Say you are a leaf group with non accounted parents. This
> > will clear the flag and so no further accounting is done. Shouldn't
> > unlimited mean that we will never reach the limit? Or am I missing
> > something?
> >
>
> You are missing something, and maybe I should be more clear about that.
> The static branches can't be disabled (it is only safe to disable them
> from disarm_static_branches(), when all references are gone). Note that
> when unlimited, we flip bits, do a transversal, but there is no mention
> to the static branch.

My little brain still doesn't get this. I wasn't concerned about static
branches. I was worried about memcg_can_account_kmem which will return
false now, doesn't it.

>
> The limiting can come and go at will.
>
> >> +
> >> + if (!memcg->use_hierarchy)
> >> +   goto out;
> >> +
> >> + for_each_mem_cgroup_tree(iter, memcg) {
> >> +   struct mem_cgroup *parent;
> >> +
> >> +   if (iter == memcg)
> >> +   continue;
> >> +   /*
> >> +   * We should only have our parent bit cleared if none
> >> +   * of our parents are accounted. The transversal order
> >> +   * of our iter function forces us to always look at the
> >> +   * parents.
> >> +   */
> >> +   parent = parent_mem_cgroup(iter);
> >> +   for (; parent != memcg; parent = parent_mem_cgroup(iter))
> >> +   if (memcg_kmem_is_accounted(parent))
> >> +   goto noclear;
> >> +   memcg_kmem_clear_account_parent(iter);
> >
> > Brain hurts...
> > Yes we are iterating in the creation ordering so we cannot rely on the
> > first encountered accounted memcg
> > A(a) - B - D
> >   - C (a) - E
>
> That's why I said I preferred the iterator the scheduler uses. The

> actual transverse code was much simpler, because it will stop at an
> unlimited parent. But this is the only drawback I see in the memcg
> iterator, so I decided that just documenting this "interesting" piece of
> code well would do...

I was just complaining that more specific comment would be much more
helpful... The ordering might be non-trivial for those who are not
familiar with cgroup internals because id doesn't tell you much.

--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v2 09/11] memcg: propagate kmem limiting information to children
Posted by Glauber Costa on Fri, 17 Aug 2012 10:07:00 GMT
View Forum Message <> Reply to Message

On 08/17/2012 01:35 PM, Michal Hocko wrote:
>>> Above you said "Once enabled, can't be disabled." and now you can
>>> > > disable it? Say you are a leaf group with non accounted parents. This
>>> > > will clear the flag and so no further accounting is done. Shouldn't
>>> > > unlimited mean that we will never reach the limit? Or am I missing
>>> > > something?
>>> > >
>> >
>> > You are missing something, and maybe I should be more clear about that.
>> > The static branches can't be disabled (it is only safe to disable them
>> > from disarm_static_branches(), when all references are gone). Note that
>> > when unlimited, we flip bits, do a transversal, but there is no mention
>> > to the static branch.
> My little brain still doesn't get this. I wasn't concerned about static
> branches. I was worried about memcg_can_account_kmem which will return
> false now, doesn't it.
>

Yes, it will. If I got you right, you are concerned because I said that
can't happen. But it will.

But I never said that can't happen. I said (ok, I meant) the static
branches can't be disabled.

Subject: Re: [PATCH v2 09/11] memcg: propagate kmem limiting information to children
Posted by Michal Hocko on Fri, 17 Aug 2012 10:35:50 GMT

On Fri 17-08-12 14:07:00, Glauber Costa wrote:
> On 08/17/2012 01:35 PM, Michal Hocko wrote:
> >>> Above you said "Once enabled, can't be disabled." and now you can
> >>> > > disable it? Say you are a leaf group with non accounted parents. This
> >>> > > will clear the flag and so no further accounting is done. Shouldn't
> >>> > > unlimited mean that we will never reach the limit? Or am I missing
> >>> > > something?
> >>> > >
> >> >
> >> > You are missing something, and maybe I should be more clear about that.
> >> > The static branches can't be disabled (it is only safe to disable them
> >> > from disarm_static_branches(), when all references are gone). Note that
> >> > when unlimited, we flip bits, do a transversal, but there is no mention
> >> > to the static branch.
> > > My little brain still doesn't get this. I wasn't concerned about static
> > > branches. I was worried about memcg_can_account_kmem which will return
> > > false now, doesn't it.
> > >
> >
> > Yes, it will. If I got you right, you are concerned because I said that
> > can't happen. But it will.
> >
> > But I never said that can't happen. I said (ok, I meant) the static
> > branches can't be disabled.

Ok, then I misunderstood that because the comment was there even before
static branches were introduced and it made sense to me. This is
inconsistent with what we do for user accounting because even if we set
limit to unlimitted we still account. Why should we differ here?

--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v2 09/11] memcg: propagate kmem limiting information to children
Posted by Glauber Costa on Fri, 17 Aug 2012 10:36:00 GMT

On 08/17/2012 02:35 PM, Michal Hocko wrote:
>> > But I never said that can't happen. I said (ok, I meant) the static
>> > branches can't be disabled.
> Ok, then I misunderstood that because the comment was there even before
> static branches were introduced and it made sense to me. This is
> inconsistent with what we do for user accounting because even if we set

> limit to unlimitted we still account. Why should we differ here?

Well, we account even without a limit for user accounting. This is a
fundamental difference, no ?

Subject: Re: [PATCH v2 09/11] memcg: propagate kmem limiting information to children
Posted by Glauber Costa on Fri, 17 Aug 2012 10:39:23 GMT
View Forum Message <> Reply to Message

On 08/17/2012 02:35 PM, Michal Hocko wrote:
> On Fri 17-08-12 14:07:00, Glauber Costa wrote:
>> On 08/17/2012 01:35 PM, Michal Hocko wrote:
>>>>> Above you said "Once enabled, can't be disabled." and now you can
>>>>>> disable it? Say you are a leaf group with non accounted parents. This
>>>>>> will clear the flag and so no further accounting is done. Shouldn't
>>>>>> unlimited mean that we will never reach the limit? Or am I missing
>>>>>> something?
>>>>>>
>>>>>
>>>>> You are missing something, and maybe I should be more clear about that.
>>>>> The static branches can't be disabled (it is only safe to disable them
>>>>> from disarm_static_branches(), when all references are gone). Note that
>>>>> when unlimited, we flip bits, do a transversal, but there is no mention
>>>>> to the static branch.
>>> My little brain still doesn't get this. I wasn't concerned about static
>>> branches. I was worried about memcg_can_account_kmem which will return
>>> false now, doesn't it.
>>>
>>
>> Yes, it will. If I got you right, you are concerned because I said that
>> can't happen. But it will.
>>
>> But I never said that can't happen. I said (ok, I meant) the static
>> branches can't be disabled.
>
> Ok, then I misunderstood that because the comment was there even before
> static branches were introduced and it made sense to me. This is
> inconsistent with what we do for user accounting because even if we set
> limit to unlimitted we still account. Why should we differ here?
>
There is another thing as well. Mel was right in his comment: I am
actually abusing this bit (because it is flippable), and it seems the
static branch can be updated more than once...

I'll merge your comments, and fix this.

Subject: Re: [PATCH v2 00/11] Request for Inclusion: kmem controller for memcg.
Posted by Ying Han on Fri, 17 Aug 2012 21:37:21 GMT
View Forum Message <> Reply to Message

On Thu, Aug 9, 2012 at 6:01 AM, Glauber Costa <glommer@parallels.com> wrote:
> 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.
>
> The slab patches are also mature in my self evaluation and could be merged not
> too long after this. For the reference, the last discussion about them happened
> at http://lwn.net/Articles/508087/
>
> A (throwaway) git tree with them is placed at:
>
> git://github.com/glommer/linux.git kmemcg-slab

I would like to make a kernel on the tree and run some perf tests on
it. However the kernel
doesn't boot due to "divide error: 0000 [#1] SMP".
https://lkml.org/lkml/2012/5/21/502

I believe the issue has been fixed (didn't look through) and can you
do a rebase on your tree?

--Ying

>
> 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 (9):
> 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
> memcg: disable kmem code when not in use.
> memcg: propagate kmem limiting information to children
> memcg: allow a memcg with kmem charges to be destructed.
> protect architectures where THREAD_SIZE >= PAGE_SIZE against fork
>   bombs
>

> Suleiman Souhlal (2):
> memcg: Make it possible to use the stock for more than one page.
> memcg: Reclaim when more than one page needed.
>
> include/linux/gfp.h | 10 +-
> include/linux/memcontrol.h | 82 ++++++++
> include/linux/thread_info.h | 2 +
> kernel/fork.c | 4 +-
> mm/memcontrol.c | 443 +++---
> mm/page_alloc.c | 38 ++++
> 6 files changed, 546 insertions(+), 33 deletions(-)
>
> --
> 1.7.11.2
>
> --
> 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

Subject: Re: [PATCH v2 00/11] Request for Inclusion: kmem controller for memcg.
Posted by Glauber Costa on Mon, 20 Aug 2012 07:51:58 GMT
View Forum Message <> Reply to Message

On 08/18/2012 01:37 AM, Ying Han wrote:
> On Thu, Aug 9, 2012 at 6:01 AM, Glauber Costa <glommer@parallels.com> wrote:
>> 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.
>>
>> The slab patches are also mature in my self evaluation and could be merged not
>> too long after this. For the reference, the last discussion about them happened
>> at http://lwn.net/Articles/508087/
>>
>> A (throwaway) git tree with them is placed at:
>>
>>   git://github.com/glommer/linux.git kmemcg-slab
>
> I would like to make a kernel on the tree and run some perf tests on
> it. However the kernel
> doesn't boot due to "divide error: 0000 [#1] SMP".

> https://lkml.org/lkml/2012/5/21/502
>
> I believe the issue has been fixed (didn't look through) and can you
> do a rebase on your tree?
>

Could you please try the branch memcg-3.5/kmemcg-slab instead? It is
rebased on top of the latest mmotm.

Subject: Re: [PATCH v2 06/11] memcg: kmem controller infrastructure
Posted by KAMEZAWA Hiroyuki on Mon, 20 Aug 2012 13:36:16 GMT
View Forum Message <> Reply to Message

(2012/08/16 2:00), Glauber Costa wrote:
> On 08/15/2012 08:38 PM, Greg Thelen wrote:
>> On Wed, Aug 15 2012, Glauber Costa wrote:
>>
>>> On 08/14/2012 10:58 PM, Greg Thelen wrote:
>>>> On Mon, Aug 13 2012, Glauber Costa wrote:
>>>>
>>>>>>>>> + WARN_ON(mem_cgroup_is_root(memcg));
>>>>>>>>> + size = (1 << order) << PAGE_SHIFT;
>>>>>>>>> + memcg_uncharge_kmem(memcg, size);
>>>>>>>>> + mem_cgroup_put(memcg);
>>>>>> Why do we need ref-counting here ? kmem res_counter cannot work as
>>>>>> reference ?
>>>>> This is of course the pair of the mem_cgroup_get() you commented on
>>>>> earlier. If we need one, we need the other. If we don't need one, we
>>>>> don't need the other =)
>>>>>
>>>>> The guarantee we're trying to give here is that the memcg structure will
>>>>> stay around while there are dangling charges to kmem, that we decided
>>>>> not to move (remember: moving it for the stack is simple, for the slab
>>>>> is very complicated and ill-defined, and I believe it is better to treat
>>>>> all kmem equally here)
>>>>
>>>> By keeping memcg structures hanging around until the last referring kmem
>>>> page is uncharged do such zombie memcg each consume a css_id and thus
>>>> put pressure on the 64k css_id space? I imagine in pathological cases
>>>> this would prevent creation of new cgroups until these zombies are
>>>> dereferenced.
>>>
>>> Yes, but although this patch makes it more likely, it doesn't introduce
>>> that. If the tasks, for instance, grab a reference to the cgroup dentry
>>> in the filesystem (like their CWD, etc), they will also keep the cgroup
>>> around.
>>

>> Fair point. But this doesn't seems like a feature. It's probably not
>> needed initially, but what do you think about creating a
>> memcg_kernel_context structure which is allocated when memcg is
>> allocated? Kernel pages charged to a memcg would have
>> page_cgroup->mem_cgroup=memcg_kernel_context rather than memcg. This
>> would allow the mem_cgroup and its css_id to be deleted when the cgroup
>> is unlinked from cgroupfs while allowing for the active kernel pages to
>> continue pointing to a valid memcg_kernel_context. This would be a
>> reference counted structure much like you are doing with memcg. When a
>> memcg is deleted the memcg_kernel_context would be linked into its
>> surviving parent memcg. This would avoid needing to visit each kernel
>> page.
>
> You need more, you need at the res_counters to stay around as well. And
> probably other fields.
>
> So my fear here is that as you add fields to that structure, you can
> defeat a bit the goal of reducing memory consumption. Still leaves the
> css space, yes. But by doing this we can introduce some subtle bugs by
> having a field in the wrong structure.
>

Hm, can't we free css_id and delete css structure from the css_id idr tree
when a memcg goes zombie ?

Thanks,
-Kame

Subject: Re: [PATCH v2 06/11] memcg: kmem controller infrastructure
Posted by Glauber Costa on Mon, 20 Aug 2012 15:29:33 GMT
View Forum Message <> Reply to Message

On 08/20/2012 05:36 PM, Kamezawa Hiroyuki wrote:
> (2012/08/16 2:00), Glauber Costa wrote:
>> On 08/15/2012 08:38 PM, Greg Thelen wrote:
>>> On Wed, Aug 15 2012, Glauber Costa wrote:
>>>
>>>> On 08/14/2012 10:58 PM, Greg Thelen wrote:
>>>>> On Mon, Aug 13 2012, Glauber Costa wrote:
>>>>>
>>>>>> +	WARN_ON(mem_cgroup_is_root(memcg));
>>>>>> +	size = (1 << order) << PAGE_SHIFT;
>>>>>> +	memcg_uncharge_kmem(memcg, size);
>>>>>> +	mem_cgroup_put(memcg);
>>>>> Why do we need ref-counting here ? kmem res_counter cannot work as
>>>>> reference ?
>>>> This is of course the pair of the mem_cgroup_get() you commented on

>>>>>> earlier. If we need one, we need the other. If we don't need one, we
>>>>>> don't need the other =)
>>>>>>
>>>>>> The guarantee we're trying to give here is that the memcg
>>>>>> structure will
>>>>>> stay around while there are dangling charges to kmem, that we decided
>>>>>> not to move (remember: moving it for the stack is simple, for the
>>>>>> slab
>>>>>> is very complicated and ill-defined, and I believe it is better to
>>>>>> treat
>>>>>> all kmem equally here)
>>>>>
>>>>> By keeping memcg structures hanging around until the last referring
>>>>> kmem
>>>>> page is uncharged do such zombie memcg each consume a css_id and thus
>>>>> put pressure on the 64k css_id space? I imagine in pathological cases
>>>>> this would prevent creation of new cgroups until these zombies are
>>>>> dereferenced.
>>>>
>>>> Yes, but although this patch makes it more likely, it doesn't introduce
>>>> that. If the tasks, for instance, grab a reference to the cgroup dentry
>>>> in the filesystem (like their CWD, etc), they will also keep the cgroup
>>>> around.
>>>
>>> Fair point. But this doesn't seems like a feature. It's probably not
>>> needed initially, but what do you think about creating a
>>> memcg_kernel_context structure which is allocated when memcg is
>>> allocated? Kernel pages charged to a memcg would have
>>> page_cgroup->mem_cgroup=memcg_kernel_context rather than memcg. This
>>> would allow the mem_cgroup and its css_id to be deleted when the cgroup
>>> is unlinked from cgroupfs while allowing for the active kernel pages to
>>> continue pointing to a valid memcg_kernel_context. This would be a
>>> reference counted structure much like you are doing with memcg. When a
>>> memcg is deleted the memcg_kernel_context would be linked into its
>>> surviving parent memcg. This would avoid needing to visit each kernel
>>> page.
>>
>> You need more, you need at the res_counters to stay around as well. And
>> probably other fields.
>>
>> So my fear here is that as you add fields to that structure, you can
>> defeat a bit the goal of reducing memory consumption. Still leaves the
>> css space, yes. But by doing this we can introduce some subtle bugs by
>> having a field in the wrong structure.
>>
>
> Hm, can't we free css_id and delete css structure from the css_id idr tree
> when a memcg goes zombie ?

>
> Thanks,
> -Kame

Kame,

I wrote a patch that does exactly that. Can you take a look? (I posted
it already)
I actually need to go back to it, because greg seems to be right saying
that that will break things for memsw. But a simplified version may work.

Subject: Re: [PATCH v2 09/11] memcg: propagate kmem limiting information to children
Posted by Michal Hocko on Tue, 21 Aug 2012 07:54:30 GMT
View Forum Message <> Reply to Message

On Fri 17-08-12 14:36:00, Glauber Costa wrote:
> On 08/17/2012 02:35 PM, Michal Hocko wrote:
> >> > But I never said that can't happen. I said (ok, I meant) the static
> >> > branches can't be disabled.
> > Ok, then I misunderstood that because the comment was there even before
> > static branches were introduced and it made sense to me. This is
> > inconsistent with what we do for user accounting because even if we set
> > limit to unlimitted we still account. Why should we differ here?
>
> Well, we account even without a limit for user accounting. This is a
> fundamental difference, no ?

Yes, user memory accounting is either on or off all the time (switchable
at boot time).
My understanding of kmem is that the feature is off by default because
it brings an overhead that is worth only special use cases. And that
sounds good to me. I do not see a good reason to have runtime switch
off. It makes the code more complicated for no good reason. E.g. how do
you handle charges you left behind? Say you charged some pages for
stack?

But maybe you have a good use case for that?
--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v2 10/11] memcg: allow a memcg with kmem charges to be destructed.
Posted by Michal Hocko on Tue, 21 Aug 2012 08:22:59 GMT

On Thu 09-08-12 17:01:18, Glauber Costa wrote:
> 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.

This changes the semantic of memory.force_empty file because the usage
should be 0 on success but it will show kmem usage in fact now. I guess
it is inevitable with u+k accounting so you should be explicit about
that and also update the documentation. If some tests (I am not 100%
sure but I guess LTP) rely on that then they could be fixed by checking
the kmem limit as well.

> Signed-off-by: Glauber Costa <glommer@parallels.com>
> 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: Johannes Weiner <hannes@cmpxchg.org>
> CC: Suleiman Souhlal <suleiman@google.com>
> ---
> mm/memcontrol.c | 17 +++++++++++++++++-
> 1 file changed, 16 insertions(+), 1 deletion(-)
>
> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
> index 3d30b79..7c1ea49 100644
> --- a/mm/memcontrol.c
> +++ b/mm/memcontrol.c
> @@ -649,6 +649,11 @@ static void disarm_kmem_keys(struct mem_cgroup *memcg)
> {
>   if (test_bit(KMEM_ACCOUNTED_THIS, &memcg->kmem_accounted))
>   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)
> @@ -4005,6 +4010,7 @@ 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);
>
> @@ -4038,8 +4044,17 @@ 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;
> --
> 1.7.11.2
>
> --
> To unsubscribe from this list: send the line "unsubscribe cgroups" in
> the body of a message to majordomo@vger.kernel.org
> More majordomo info at http://vger.kernel.org/majordomo-info.html

--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v2 09/11] memcg: propagate kmem limiting information to children

Posted by Michal Hocko on Tue, 21 Aug 2012 08:35:01 GMT

View Forum Message <> Reply to Message

On Tue 21-08-12 09:54:30, Michal Hocko wrote:
> E.g. how do you handle charges you left behind? Say you charged some
> pages for stack?

I got to the last patch and see how you do it. You are relying on
free_accounted_pages directly which doesn't check kmem_accounted and
uses PageUsed bit instead. So this is correct. I guess you are relying
on the life cycle of the object in general so other types of objects
should be safe as well and there shouldn't be any leaks. It is just that
the memcg life time is not bounded now. Will think about that.
--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v2 09/11] memcg: propagate kmem limiting information to children
Posted by Glauber Costa on Tue, 21 Aug 2012 09:17:14 GMT
View Forum Message <> Reply to Message

On 08/21/2012 12:35 PM, Michal Hocko wrote:
> On Tue 21-08-12 09:54:30, Michal Hocko wrote:
>> E.g. how do you handle charges you left behind? Say you charged some
>> pages for stack?
>
> I got to the last patch and see how you do it. You are relying on
> free_accounted_pages directly which doesn't check kmem_accounted and
> uses PageUsed bit instead. So this is correct. I guess you are relying
> on the life cycle of the object in general so other types of objects
> should be safe as well and there shouldn't be any leaks. It is just that
> the memcg life time is not bounded now. Will think about that.
>
Unless you have a better way, I believe any kind of transversal in the
free page path is performance detrimental. So the best way is to be
explicit and mark a specific callsite as a memcg free.

As for the unbounded time, you are correct. However, I believe it is
possible to move a lot of the work we do for free (such as freeing the
percpu counters and the css_id itself) to an earlier time.

Also, if it ever becomes a problem, it is theoretically possible to
avoid this, by tracking the kmem pages in a per-memcg list. We would
then transverse such list as we do for user pages, and reparent them.
The problem is that this is also a bit space inefficient, since we can't
reuse any more fields in page_struct for the list_head, so we'd need an
external structure. There is a list_head + a pointer per tracked page.

Subject: Re: [PATCH v2 09/11] memcg: propagate kmem limiting information to children
Posted by Glauber Costa on Tue, 21 Aug 2012 09:22:09 GMT

On 08/21/2012 11:54 AM, Michal Hocko wrote:
> On Fri 17-08-12 14:36:00, Glauber Costa wrote:
>> On 08/17/2012 02:35 PM, Michal Hocko wrote:
>>>>> But I never said that can't happen. I said (ok, I meant) the static
>>>>> branches can't be disabled.
>>> Ok, then I misunderstood that because the comment was there even before
>>> static branches were introduced and it made sense to me. This is
>>> inconsistent with what we do for user accounting because even if we set
>>> limit to unlimitted we still account. Why should we differ here?
>>
>> Well, we account even without a limit for user accounting. This is a
>> fundamental difference, no ?
>
> Yes, user memory accounting is either on or off all the time (switchable
> at boot time).
> My understanding of kmem is that the feature is off by default because
> it brings an overhead that is worth only special use cases. And that
> sounds good to me. I do not see a good reason to have runtime switch
> off. It makes the code more complicated for no good reason. E.g. how do
> you handle charges you left behind? Say you charged some pages for
> stack?
>
Answered in your other e-mail. About the code complication, yes, it does
make the code more complicated. See below.

> But maybe you have a good use case for that?
>
Honestly, I don't. For my particular use case, this would be always on,
and end of story. I was operating under the belief that being able to
say "Oh, I regret", and then turning it off would be beneficial, even at
the expense of the - self contained - complication.

For the general sanity of the interface, it is also a bit simpler to say
"if kmem is unlimited, x happens", which is a verifiable statement, than
to have a statement that is dependent on past history. But all of those
need of course, as you pointed out, to be traded off by the code complexity.

I am fine with either, I just need a clear sign from you guys so I don't
keep deimplementing and reimplementing this forever.

Subject: Re: [PATCH v2 11/11] protect architectures where THREAD_SIZE >= PAGE_SIZE against fork bombs
Posted by Michal Hocko on Tue, 21 Aug 2012 09:35:13 GMT

On Thu 09-08-12 17:01:19, Glauber Costa wrote:
> 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.

quick git grep "define *THREAD_SIZE\>" arch says that there is no such
architecture.

> 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 :(){ :|:& };:

I guess there were no other tasks in the same group (except for the
parent shell), right? I am asking because this should trigger memcg-oom
but that one will usually pick up something else than the fork bomb
which would have a small memory footprint. But that needs to be handled
on the oom level obviously.

> Signed-off-by: Glauber Costa <glommer@parallels.com>
> Acked-by: Frederic Weisbecker <fweisbec@redhat.com>
> CC: Christoph Lameter <cl@linux.com>
> CC: Pekka Enberg <penberg@cs.helsinki.fi>
> CC: Michal Hocko <mhocko@suse.cz>
> CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
> CC: Johannes Weiner <hannes@cmpxchg.org>
> CC: Suleiman Souhlal <suleiman@google.com>

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

> ---
> 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 dc3ff16..b0b90c3 100644
> --- a/kernel/fork.c
> +++ b/kernel/fork.c
> @@ -142,7 +142,7 @@ 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;
> @@ -151,7 +151,7 @@ static struct thread_info *alloc_thread_info_node(struct task_struct
*tsk,
> static inline void free_thread_info(struct thread_info *ti)
> {
>   arch_release_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;
> --
> 1.7.11.2
>
> --
> To unsubscribe from this list: send the line "unsubscribe cgroups" in
> the body of a message to majordomo@vger.kernel.org
> More majordomo info at http://vger.kernel.org/majordomo-info.html

--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v2 11/11] protect architectures where THREAD_SIZE >=

PAGE_SIZE against fork bombs
Posted by Glauber Costa on Tue, 21 Aug 2012 09:40:45 GMT
View Forum Message <> Reply to Message

On 08/21/2012 01:35 PM, Michal Hocko wrote:
> On Thu 09-08-12 17:01:19, Glauber Costa wrote:
>> 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.
>
> quick git grep "define *THREAD_SIZE\>" arch says that there is no such
> architecture.
>
>> 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 :(){ :|:& };:
>
> I guess there were no other tasks in the same group (except for the
> parent shell), right?

Yes.

> I am asking because this should trigger memcg-oom
> but that one will usually pick up something else than the fork bomb
> which would have a small memory footprint. But that needs to be handled
> on the oom level obviously.
>
Sure, but keep in mind that the main protection is against tasks *not*
in this memcg.

Subject: Re: [PATCH v2 09/11] memcg: propagate kmem limiting information to children
Posted by Michal Hocko on Tue, 21 Aug 2012 10:00:07 GMT
View Forum Message <> Reply to Message

On Tue 21-08-12 13:22:09, Glauber Costa wrote:

> On 08/21/2012 11:54 AM, Michal Hocko wrote:
[...]
> > But maybe you have a good use case for that?
> >
> Honestly, I don't. For my particular use case, this would be always on,
> and end of story. I was operating under the belief that being able to
> say "Oh, I regret", and then turning it off would be beneficial, even at
> the expense of the - self contained - complication.
>
> For the general sanity of the interface, it is also a bit simpler to say
> "if kmem is unlimited, x happens", which is a verifiable statement, than
> to have a statement that is dependent on past history.

OK, fair point. We shouldn't rely on the history. Maybe
memory.kmem.limit_in_bytes could return some special value like -1 in
such a case?

> But all of those need of course, as you pointed out, to be traded off
> by the code complexity.
>
> I am fine with either, I just need a clear sign from you guys so I don't
> keep deimplementing and reimplementing this forever.

I would be for make it simple now and go with additional features later
when there is a demand for them. Maybe we will have runtimg switch for
user memory accounting as well one day.

But let's see what others think?
--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v2 09/11] memcg: propagate kmem limiting information to children
Posted by Glauber Costa on Tue, 21 Aug 2012 10:01:24 GMT
View Forum Message <> Reply to Message

On 08/21/2012 02:00 PM, Michal Hocko wrote:
> On Tue 21-08-12 13:22:09, Glauber Costa wrote:
>> On 08/21/2012 11:54 AM, Michal Hocko wrote:
> [...]
>>> But maybe you have a good use case for that?
>>>
>> Honestly, I don't. For my particular use case, this would be always on,
>> and end of story. I was operating under the belief that being able to
>> say "Oh, I regret", and then turning it off would be beneficial, even at
>> the expense of the - self contained - complication.

>>
>> For the general sanity of the interface, it is also a bit simpler to say
>> "if kmem is unlimited, x happens", which is a verifiable statement, than
>> to have a statement that is dependent on past history.
>
> OK, fair point. We shouldn't rely on the history. Maybe
> memory.kmem.limit_in_bytes could return some special value like -1 in
> such a case?
>

Way I see it, this is simplifying the code at the expense of
complicating the interface.

>> But all of those need of course, as you pointed out, to be traded off
>> by the code complexity.
>>
>> I am fine with either, I just need a clear sign from you guys so I don't
>> keep deimplementing and reimplementing this forever.
>
> I would be for make it simple now and go with additional features later
> when there is a demand for them. Maybe we will have runtimg switch for
> user memory accounting as well one day.
>

Since this would change a then established behavior, the same
discussions about compatibility we ever get to will rise. It is a pain
we'd better avoid if we can.

> But let's see what others think?
>

Absolutely. Hello others, what do you think ?

Subject: Re: [PATCH v2 11/11] protect architectures where THREAD_SIZE >= PAGE_SIZE against fork bombs
Posted by Michal Hocko on Tue, 21 Aug 2012 10:57:04 GMT
View Forum Message <> Reply to Message

On Tue 21-08-12 13:40:45, Glauber Costa wrote:
> On 08/21/2012 01:35 PM, Michal Hocko wrote:
[...]
> > I am asking because this should trigger memcg-oom
> > but that one will usually pick up something else than the fork bomb
> > which would have a small memory footprint. But that needs to be handled
> > on the oom level obviously.
> >
> Sure, but keep in mind that the main protection is against tasks *not*

> in this memcg.

Yes and that's is good step forward. I just wanted to mention that we
still have the problem inside the subhierarchy. The changelog was not
specific enough.
--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v2 06/11] memcg: kmem controller infrastructure
Posted by Greg Thelen on Tue, 21 Aug 2012 21:50:54 GMT
View Forum Message <> Reply to Message

On Thu, Aug 09 2012, 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 accessors.  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.
>
> [ v2: improved comments and standardized function names ]
>
> Signed-off-by: Glauber Costa <glommer@parallels.com>
> CC: Christoph Lameter <cl@linux.com>
> CC: Pekka Enberg <penberg@cs.helsinki.fi>
> CC: Michal Hocko <mhocko@suse.cz>
> CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
> CC: Johannes Weiner <hannes@cmpxchg.org>
> ---
>  include/linux/memcontrol.h |  79 +++++++++++++++++++
>  mm/memcontrol.c            | 185 ++++++++++++++++++++++++++++++++++++++++++++++++++++++
>  2 files changed, 264 insertions(+)
>
> diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
> index 8d9489f..75b247e 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 +400,11 @@ struct sock;
> #ifdef CONFIG_MEMCG_KMEM
> void sock_update_memcg(struct sock *sk);
> void sock_release_memcg(struct sock *sk);
> +
> +#define memcg_kmem_on 1
> +bool __memcg_kmem_new_page(gfp_t gfp, void *handle, int order);
> +void __memcg_kmem_commit_page(struct page *page, void *handle, int order);
> +void __memcg_kmem_free_page(struct page *page, int order);
> #else
> static inline void sock_update_memcg(struct sock *sk)
> {
> @@ -406,6 +412,79 @@ static inline void sock_update_memcg(struct sock *sk)
> static inline void sock_release_memcg(struct sock *sk)
> {
> }
> +
> +#define memcg_kmem_on 0
> +static inline bool
> +__memcg_kmem_new_page(gfp_t gfp, void *handle, int order)
> +{
> + return false;
> +}
> +
> +static inline void __memcg_kmem_free_page(struct page *page, int order)
> +{
> +}
> +
> +static inline void
> +__memcg_kmem_commit_page(struct page *page, struct mem_cgroup *handle, int order)
> +{
> +}
> #endif /* CONFIG_MEMCG_KMEM */
> +
> +/**
> + * memcg_kmem_new_page: verify if a new kmem allocation is allowed.
> + * @gfp: the gfp allocation flags.
> + * @handle: 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_new_page(gfp_t gfp, void *handle, int order)
> +{
> + if (!memcg_kmem_on)
> + return true;
> + if (!(gfp & __GFP_KMEMCG) || (gfp & __GFP_NOFAIL))
> + return true;
> + if (in_interrupt() || (!current->mm) || (current->flags & PF_KTHREAD))
> + return true;
> + return __memcg_kmem_new_page(gfp, handle, order);
> +}
> +
> +/**
> + * memcg_kmem_free_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_free_page(struct page *page, int order)
> +{
> + if (memcg_kmem_on)
> + __memcg_kmem_free_page(page, order);
> +}
> +
> +/**
> + * memcg_kmem_commit_page: embeds correct memcg in a page
> + * @handle: a pointer to the memcg this was charged against.
> + * @page: pointer to struct page recently allocated
> + * @handle: the memcg structure we charged against
> + * @order: allocation order.
> + *
> + * Needs to be called after memcg_kmem_new_page, 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 @handle to the
> + * corresponding page_cgroup.
> + */
> +static __always_inline void
> +memcg_kmem_commit_page(struct page *page, struct mem_cgroup *handle, int order)
> +{
> + if (memcg_kmem_on)
> + __memcg_kmem_commit_page(page, handle, order);
> +}
> #endif /* _LINUX_MEMCONTROL_H */
>
> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
> index 54e93de..e9824c1 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
> @@ -434,6 +438,9 @@ struct mem_cgroup *mem_cgroup_from_css(struct
cgroup_subsys_state *s)
> #include <net/ip.h>
>
> static bool mem_cgroup_is_root(struct mem_cgroup *memcg);
> +static int memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t gfp, s64 delta);
> +static void memcg_uncharge_kmem(struct mem_cgroup *memcg, s64 delta);
> +
> void sock_update_memcg(struct sock *sk)
> {
>   if (mem_cgroup_sockets_enabled) {
> @@ -488,6 +495,118 @@ struct cg_proto *tcp_proto_cgroup(struct mem_cgroup *memcg)
> }
> EXPORT_SYMBOL(tcp_proto_cgroup);
> #endif /* CONFIG_INET */
> +
> +static inline bool memcg_kmem_enabled(struct mem_cgroup *memcg)
> +{
> + return !mem_cgroup_disabled() && !mem_cgroup_is_root(memcg) &&
> + memcg->kmem_accounted;
> +}
> +
> +/*
> + * 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
> + *
> + * Returning true means the allocation is possible.
> + */
> +bool __memcg_kmem_new_page(gfp_t gfp, void *_handle, int order)
> +{

```
> + struct mem_cgroup *memcg;
> + struct mem_cgroup **handle = (struct mem_cgroup **)_handle;
> + bool ret = true;
> + size_t size;
> + struct task_struct *p;
> +
> + *handle = NULL;
> + rcu_read_lock();
> + p = rcu_dereference(current->mm->owner);
> + memcg = mem_cgroup_from_task(p);
> + if (!memcg_kmem_enabled(memcg))
> +  goto out;
> +
> + mem_cgroup_get(memcg);
> +
> + size = PAGE_SIZE << order;
> + ret = memcg_charge_kmem(memcg, gfp, size) == 0;
> + if (!ret) {
> +  mem_cgroup_put(memcg);
> +  goto out;
> + }
> +
> + *handle = memcg;
> +out:
> + rcu_read_unlock();
> + return ret;
> +}
> +EXPORT_SYMBOL(__memcg_kmem_new_page);
> +
> +void __memcg_kmem_commit_page(struct page *page, void *handle, int order)
> +{
> + struct page_cgroup *pc;
> + struct mem_cgroup *memcg = handle;
> +
> + if (!memcg)
> +  return;
> +
> + WARN_ON(mem_cgroup_is_root(memcg));
> + /* The page allocation must have failed. Revert */
> + if (!page) {
> +  size_t size = PAGE_SIZE << order;
> +
> +  memcg_uncharge_kmem(memcg, size);
> +  mem_cgroup_put(memcg);
> +  return;

> +
> + pc = lookup_page_cgroup(page);
```

> + lock_page_cgroup(pc);
> + pc->mem_cgroup = memcg;
> + SetPageCgroupUsed(pc);
> + unlock_page_cgroup(pc);

I have no problem with the code here. But, out of curiosity, why do we need to lock the pc here and below in __memcg_kmem_free_page()?

For the allocating side, I don't think that migration or reclaim will be manipulating this page. But is there something else that we need the locking for?

For the freeing side, it seems that anyone calling __memcg_kmem_free_page() is going to be freeing a previously accounted page.

I imagine that if we did not need the locking we would still need some memory barriers to make sure that modifications to the PG_lru are serialized wrt. to kmem modifying PageCgroupUsed here.

Perhaps we're just trying to take a conservative initial implementation which is consistent with user visible pages.

> +}
> +
> +void __memcg_kmem_free_page(struct page *page, int order)
> +{
> + struct mem_cgroup *memcg;
> + size_t size;
> + struct page_cgroup *pc;
> +
> + if (mem_cgroup_disabled())
> + return;
> +
> + pc = lookup_page_cgroup(page);
> + lock_page_cgroup(pc);
> + memcg = pc->mem_cgroup;
> + pc->mem_cgroup = NULL;
> + if (!PageCgroupUsed(pc)) {

When do we expect to find PageCgroupUsed() unset in this routine? Is this just to handle the race of someone enabling kmem accounting after allocating a page and then later freeing that page?

> + unlock_page_cgroup(pc);
> + return;
> + }
> + ClearPageCgroupUsed(pc);

> + unlock_page_cgroup(pc);
> +
> + /*
> + * Checking if kmem accounted is enabled won't work for uncharge, since
> + * it is possible that the user enabled kmem tracking, allocated, and
> + * then disabled it again.
> + *
> + * We trust if there is a memcg associated with the page, it is a valid
> + * allocation
> + */
> + if (!memcg)
> + return;
> +
> + WARN_ON(mem_cgroup_is_root(memcg));
> + size = (1 << order) << PAGE_SHIFT;
> + memcg_uncharge_kmem(memcg, size);
> + mem_cgroup_put(memcg);
> +}
> +EXPORT_SYMBOL(__memcg_kmem_free_page);
> #endif /* CONFIG_MEMCG_KMEM */
>
> #if defined(CONFIG_INET) && defined(CONFIG_MEMCG_KMEM)
> @@ -5759,3 +5878,69 @@ static int __init enable_swap_account(char *s)
> __setup("swapaccount=", enable_swap_account);
>
> #endif
> +
> +#ifdef CONFIG_MEMCG_KMEM
> +int memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t gfp, s64 delta)
> +{
> + struct res_counter *fail_res;
> + struct mem_cgroup *_memcg;
> + int ret;
> + bool may_oom;
> + bool nofail = false;
> +
> + may_oom = (gfp & __GFP_WAIT) && (gfp & __GFP_FS) &&
> +   !(gfp & __GFP_NORETRY);
> +
> + ret = 0;
> +
> + if (!memcg)
> + return ret;
> +
> + _memcg = memcg;
> + ret = __mem_cgroup_try_charge(NULL, gfp, delta / PAGE_SIZE,
> +   &_memcg, may_oom);
> +

```
> + if (ret == -EINTR)  {
> +  nofail = true;
> +  /*
> +   * __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
> +   */
> +  res_counter_charge_nofail(&memcg->res, delta, &fail_res);
> +  if (do_swap_account)
> +   res_counter_charge_nofail(&memcg->memsw, delta,
> +         &fail_res);
> +  ret = 0;
> + } else if (ret == -ENOMEM)
> +  return ret;
> +
> + if (nofail)
> +  res_counter_charge_nofail(&memcg->kmem, delta, &fail_res);
> + else
> +  ret = res_counter_charge(&memcg->kmem, delta, &fail_res);
> +
> + if (ret) {
> +  res_counter_uncharge(&memcg->res, delta);
> +  if (do_swap_account)
> +   res_counter_uncharge(&memcg->memsw, delta);
> + }
> +
> + return ret;
> +}
> +
> +void memcg_uncharge_kmem(struct mem_cgroup *memcg, s64 delta)
> +{
> + if (!memcg)
> +  return;
> +
> + res_counter_uncharge(&memcg->kmem, delta);
> + res_counter_uncharge(&memcg->res, delta);
> + if (do_swap_account)
> +  res_counter_uncharge(&memcg->memsw, delta);
> +}
> +#endif /* CONFIG_MEMCG_KMEM */
```

Subject: Re: [PATCH v2 09/11] memcg: propagate kmem limiting information to

On Tue, Aug 21 2012, Michal Hocko wrote:

> On Tue 21-08-12 13:22:09, Glauber Costa wrote:
>> On 08/21/2012 11:54 AM, Michal Hocko wrote:
> [...]
>> > But maybe you have a good use case for that?
>> >
>> Honestly, I don't. For my particular use case, this would be always on,
>> and end of story. I was operating under the belief that being able to
>> say "Oh, I regret", and then turning it off would be beneficial, even at
>> the expense of the - self contained - complication.
>>
>> For the general sanity of the interface, it is also a bit simpler to say
>> "if kmem is unlimited, x happens", which is a verifiable statement, than
>> to have a statement that is dependent on past history.
>
> OK, fair point. We shouldn't rely on the history. Maybe
> memory.kmem.limit_in_bytes could return some special value like -1 in
> such a case?
>
>> But all of those need of course, as you pointed out, to be traded off
>> by the code complexity.
>>
>> I am fine with either, I just need a clear sign from you guys so I don't
>> keep deimplementing and reimplementing this forever.
>
> I would be for make it simple now and go with additional features later
> when there is a demand for them. Maybe we will have runtimg switch for
> user memory accounting as well one day.
>
> But let's see what others think?

In my use case memcg will either be disable or (enabled and kmem
limiting enabled).

I'm not sure I follow the discussion about history. Are we saying that
once a kmem limit is set then kmem will be accounted/charged to memcg.
Is this discussion about the static branches/etc that are autotuned the
first time is enabled? The first time its set there parts of the system
will be adjusted in such a way that may impose a performance overhead
(static branches, etc). Thereafter the performance cannot be regained
without a reboot. This makes sense to me. Are we saying that
kmem.limit_in_bytes will have three states?
- kmem never enabled on machine therefore kmem has never been enabled
- kmem has been enabled in past but is not effective is this cgroup

(limit=infinity)
- kmem is effective in this mem (limit=not-infinity)

Subject: Re: [PATCH v2 09/11] memcg: propagate kmem limiting information to children
Posted by Glauber Costa on Wed, 22 Aug 2012 08:22:49 GMT
View Forum Message <> Reply to Message

>>>
>>> I am fine with either, I just need a clear sign from you guys so I don't
>>> keep deimplementing and reimplementing this forever.
>>
>> I would be for make it simple now and go with additional features later
>> when there is a demand for them. Maybe we will have runtimg switch for
>> user memory accounting as well one day.
>>
>> But let's see what others think?
>
> In my use case memcg will either be disable or (enabled and kmem
> limiting enabled).
>
> I'm not sure I follow the discussion about history. Are we saying that
> once a kmem limit is set then kmem will be accounted/charged to memcg.
> Is this discussion about the static branches/etc that are autotuned the
> first time is enabled?

No, the question is about when you unlimit a former kmem-limited memcg.

> The first time its set there parts of the system
> will be adjusted in such a way that may impose a performance overhead
> (static branches, etc). Thereafter the performance cannot be regained
> without a reboot. This makes sense to me. Are we saying that
> kmem.limit_in_bytes will have three states?

It is not about performance, about interface.

Michal says that once a particular memcg was kmem-limited, it will keep
accounting pages, even if you make it unlimited. The limits won't be
enforced, for sure - there is no limit, but pages will still be accounted.

This simplifies the code galore, but I worry about the interface: A
person looking at the current status of the files only, without
knowledge of past history, can't tell if allocations will be tracked or not.

Subject: Re: [PATCH v2 06/11] memcg: kmem controller infrastructure

Posted by Glauber Costa on Wed, 22 Aug 2012 08:35:24 GMT
View Forum Message <> Reply to Message

On 08/22/2012 01:50 AM, Greg Thelen wrote:
> On Thu, Aug 09 2012, 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 accessors. 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.
>>
>> [v2: improved comments and standardized function names]
>>
>> Signed-off-by: Glauber Costa <glommer@parallels.com>
>> CC: Christoph Lameter <cl@linux.com>
>> CC: Pekka Enberg <penberg@cs.helsinki.fi>
>> CC: Michal Hocko <mhocko@suse.cz>
>> CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
>> CC: Johannes Weiner <hannes@cmpxchg.org>
>> ---
>> include/linux/memcontrol.h | 79 +++++++++++++++++++
>> mm/memcontrol.c | 185 +++
>> 2 files changed, 264 insertions(+)
>>
>> diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
>> index 8d9489f..75b247e 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 +400,11 @@ struct sock;
>> #ifdef CONFIG_MEMCG_KMEM
>> void sock_update_memcg(struct sock *sk);
>> void sock_release_memcg(struct sock *sk);
>> +
>> +#define memcg_kmem_on 1
>> +bool __memcg_kmem_new_page(gfp_t gfp, void *handle, int order);
>> +void __memcg_kmem_commit_page(struct page *page, void *handle, int order);
>> +void __memcg_kmem_free_page(struct page *page, int order);
>> #else

>> static inline void sock_update_memcg(struct sock *sk)
>> {
>> @@ -406,6 +412,79 @@ static inline void sock_update_memcg(struct sock *sk)
>> static inline void sock_release_memcg(struct sock *sk)
>> {
>> }
>> +
>> +#define memcg_kmem_on 0
>> +static inline bool
>> +__memcg_kmem_new_page(gfp_t gfp, void *handle, int order)
>> +{
>> + return false;
>> +}
>> +
>> +static inline void __memcg_kmem_free_page(struct page *page, int order)
>> +{
>> +}
>> +
>> +static inline void
>> +__memcg_kmem_commit_page(struct page *page, struct mem_cgroup *handle, int order)
>> +{
>> +}
>> #endif /* CONFIG_MEMCG_KMEM */
>> +
>> +/**
>> + * memcg_kmem_new_page: verify if a new kmem allocation is allowed.
>> + * @gfp: the gfp allocation flags.
>> + * @handle: 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_new_page(gfp_t gfp, void *handle, int order)
>> +{
>> + if (!memcg_kmem_on)
>> + return true;
>> + if (!(gfp & __GFP_KMEMCG) || (gfp & __GFP_NOFAIL))
>> + return true;
>> + if (in_interrupt() || (!current->mm) || (current->flags & PF_KTHREAD))
>> + return true;
>> + return __memcg_kmem_new_page(gfp, handle, order);
>> +}
>> +

```
>> +/**
>> + * memcg_kmem_free_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_free_page(struct page *page, int order)
>> +{
>> + if (memcg_kmem_on)
>> +  __memcg_kmem_free_page(page, order);
>> +}
>> +
>> +/**
>> + * memcg_kmem_commit_page: embeds correct memcg in a page
>> + * @handle: a pointer to the memcg this was charged against.
>> + * @page: pointer to struct page recently allocated
>> + * @handle: the memcg structure we charged against
>> + * @order: allocation order.
>> + *
>> + * Needs to be called after memcg_kmem_new_page, 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 @handle to the
>> + * corresponding page_cgroup.
>> + */
>> +static __always_inline void
>> +memcg_kmem_commit_page(struct page *page, struct mem_cgroup *handle, int order)
>> +{
>> + if (memcg_kmem_on)
>> +  __memcg_kmem_commit_page(page, handle, order);
>> +}
>>  #endif /* _LINUX_MEMCONTROL_H */
>>
>> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
>> index 54e93de..e9824c1 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
>> @@ -434,6 +438,9 @@ struct mem_cgroup *mem_cgroup_from_css(struct
cgroup_subsys_state *s)
>> #include <net/ip.h>
>>
>> static bool mem_cgroup_is_root(struct mem_cgroup *memcg);
>> +static int memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t gfp, s64 delta);
>> +static void memcg_uncharge_kmem(struct mem_cgroup *memcg, s64 delta);
>> +
>> void sock_update_memcg(struct sock *sk)
>> {
>>   if (mem_cgroup_sockets_enabled) {
>> @@ -488,6 +495,118 @@ struct cg_proto *tcp_proto_cgroup(struct mem_cgroup *memcg)
>> }
>> EXPORT_SYMBOL(tcp_proto_cgroup);
>> #endif /* CONFIG_INET */
>> +
>> +static inline bool memcg_kmem_enabled(struct mem_cgroup *memcg)
>> +{
>> + return !mem_cgroup_disabled() && !mem_cgroup_is_root(memcg) &&
>> + memcg->kmem_accounted;
>> +}
>> +
>> +/*
>> + * 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
>> + *
>> + * Returning true means the allocation is possible.
>> + */
>> +bool __memcg_kmem_new_page(gfp_t gfp, void *_handle, int order)
>> +{
>> + struct mem_cgroup *memcg;
>> + struct mem_cgroup **handle = (struct mem_cgroup **)_handle;
>> + bool ret = true;
>> + size_t size;
>> + struct task_struct *p;
>> +
>> + *handle = NULL;
>> + rcu_read_lock();
>> + p = rcu_dereference(current->mm->owner);
>> + memcg = mem_cgroup_from_task(p);
>> + if (!memcg_kmem_enabled(memcg))

>> + goto out;
>> +
>> + mem_cgroup_get(memcg);
>> +
>> + size = PAGE_SIZE << order;
>> + ret = memcg_charge_kmem(memcg, gfp, size) == 0;
>> + if (!ret) {
>> + mem_cgroup_put(memcg);
>> + goto out;
>> + }
>> +
>> + *handle = memcg;
>> +out:
>> + rcu_read_unlock();
>> + return ret;
>> +}
>> +EXPORT_SYMBOL(__memcg_kmem_new_page);
>> +
>> +void __memcg_kmem_commit_page(struct page *page, void *handle, int order)
>> +{
>> + struct page_cgroup *pc;
>> + struct mem_cgroup *memcg = handle;
>> +
>> + if (!memcg)
>> + return;
>> +
>> + WARN_ON(mem_cgroup_is_root(memcg));
>> + /* The page allocation must have failed. Revert */
>> + if (!page) {
>> + size_t size = PAGE_SIZE << order;
>> +
>> + memcg_uncharge_kmem(memcg, size);
>> + mem_cgroup_put(memcg);
>> + return;
>
>> +
>> + pc = lookup_page_cgroup(page);
>> + lock_page_cgroup(pc);
>> + pc->mem_cgroup = memcg;
>> + SetPageCgroupUsed(pc);
>> + unlock_page_cgroup(pc);
>
> I have no problem with the code here. But, out of curiosity, why do we
> need to lock the pc here and below in __memcg_kmem_free_page()?
>
> For the allocating side, I don't think that migration or reclaim will be
> manipulating this page. But is there something else that we need the
> locking for?

>
> For the freeing side, it seems that anyone calling
> __memcg_kmem_free_page() is going to be freeing a previously accounted
> page.
>
> I imagine that if we did not need the locking we would still need some
> memory barriers to make sure that modifications to the PG_lru are
> serialized wrt. to kmem modifying PageCgroupUsed here.
>
Unlocking should do that, no?

> Perhaps we're just trying to take a conservative initial implementation
> which is consistent with user visible pages.
>

The way I see it, is not about being conservative, but rather about my physical safety. It is quite easy and natural to assume that "all modifications to page cgroup are done under lock". So someone modifying this later will likely find out about this exception in a rather unpleasant way. They know where I live, and guns for hire are everywhere.

Note that it is not unreasonable to believe that we can modify this later. This can be a way out, for example, for the memcg lifecycle problem.

I agree with your analysis and we can ultimately remove it, but if we cannot pinpoint any performance problems to here, maybe consistency wins. Also, the locking operation itself is a bit expensive, but the biggest price is the actual contention. If we'll have nobody contending for the same page_cgroup, the problem - if exists - shouldn't be that bad. And if we ever have, the lock is needed.

>> +}
>> +
>> +void __memcg_kmem_free_page(struct page *page, int order)
>> +{
>> + struct mem_cgroup *memcg;
>> + size_t size;
>> + struct page_cgroup *pc;
>> +
>> + if (mem_cgroup_disabled())
>> + return;
>> +
>> + pc = lookup_page_cgroup(page);
>> + lock_page_cgroup(pc);
>> + memcg = pc->mem_cgroup;
>> + pc->mem_cgroup = NULL;
>> + if (!PageCgroupUsed(pc)) {

>
> When do we expect to find PageCgroupUsed() unset in this routine? Is
> this just to handle the race of someone enabling kmem accounting after
> allocating a page and then later freeing that page?
>

All the time we have a valid memcg. It is marked Used at charge time, so this is how we differentiate between a tracked page and a non-tracked page. Note that even though we explicit mark the freeing call sites with free_allocated_page, etc, not all pc->memcg will be valid. There are unlimited memcgs, bypassed charges, GFP_NOFAIL allocations, etc.

Subject: Re: [PATCH v2 10/11] memcg: allow a memcg with kmem charges to be destructed.
Posted by Glauber Costa on Wed, 22 Aug 2012 08:36:17 GMT
View Forum Message <> Reply to Message

On 08/21/2012 12:22 PM, Michal Hocko wrote:
> On Thu 09-08-12 17:01:18, Glauber Costa wrote:
>> 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.
>
> This changes the semantic of memory.force_empty file because the usage
> should be 0 on success but it will show kmem usage in fact now. I guess
> it is inevitable with u+k accounting so you should be explicit about
> that and also update the documentation.
aaand, it's done.

Subject: Re: [PATCH v2 09/11] memcg: propagate kmem limiting information to children
Posted by Greg Thelen on Wed, 22 Aug 2012 23:23:12 GMT
View Forum Message <> Reply to Message

On Wed, Aug 22 2012, Glauber Costa wrote:

>>>>
>>>> I am fine with either, I just need a clear sign from you guys so I don't
>>>> keep deimplementing and reimplementing this forever.

>>>
>>> I would be for make it simple now and go with additional features later
>>> when there is a demand for them. Maybe we will have runtimg switch for
>>> user memory accounting as well one day.
>>>
>>> But let's see what others think?
>>
>> In my use case memcg will either be disable or (enabled and kmem
>> limiting enabled).
>>
>> I'm not sure I follow the discussion about history. Are we saying that
>> once a kmem limit is set then kmem will be accounted/charged to memcg.
>> Is this discussion about the static branches/etc that are autotuned the
>> first time is enabled?
>
> No, the question is about when you unlimit a former kmem-limited memcg.
>
>> The first time its set there parts of the system
>> will be adjusted in such a way that may impose a performance overhead
>> (static branches, etc). Thereafter the performance cannot be regained
>> without a reboot. This makes sense to me. Are we saying that
>> kmem.limit_in_bytes will have three states?
>
> It is not about performance, about interface.
>
> Michal says that once a particular memcg was kmem-limited, it will keep
> accounting pages, even if you make it unlimited. The limits won't be
> enforced, for sure - there is no limit, but pages will still be accounted.
>
> This simplifies the code galore, but I worry about the interface: A
> person looking at the current status of the files only, without
> knowledge of past history, can't tell if allocations will be tracked or not.

In the current patch set we've conflating enabling kmem accounting with
the kmem limit value (RESOURCE_MAX=disabled, all_other_values=enabled).

I see no problem with simpling the kernel code with the requirement that
once a particular memcg enables kmem accounting that it cannot be
disabled for that memcg.

The only question is the user space interface. Two options spring to
mind:
a) Close to current code. Once kmem.limit_in_bytes is set to
   non-RESOURCE_MAX, then kmem accounting is enabled and cannot be
   disabled. Therefore the limit cannot be set to RESOURCE_MAX
   thereafter. The largest value would be something like
   RESOURCE_MAX-PAGE_SIZE. An admin wondering if kmem is enabled only
   has to cat kmem.limit_in_bytes - if it's less than RESOURCE_MAX, then

kmem is enabled.

b) Or, if we could introduce a separate sticky kmem.enabled file. Once
set it could not be unset. Kmem accounting would only be enabled if
kmem.enabled=1.

I think (b) is clearer.

Subject: Re: [PATCH v2 06/11] memcg: kmem controller infrastructure
Posted by Greg Thelen on Thu, 23 Aug 2012 00:07:45 GMT
View Forum Message <> Reply to Message

On Wed, Aug 22 2012, Glauber Costa wrote:

> On 08/22/2012 01:50 AM, Greg Thelen wrote:
>> On Thu, Aug 09 2012, 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 accessors. 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.
>>>
>>> [v2: improved comments and standardized function names]
>>>
>>> Signed-off-by: Glauber Costa <glommer@parallels.com>
>>> CC: Christoph Lameter <cl@linux.com>
>>> CC: Pekka Enberg <penberg@cs.helsinki.fi>
>>> CC: Michal Hocko <mhocko@suse.cz>
>>> CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
>>> CC: Johannes Weiner <hannes@cmpxchg.org>
>>> ---
>>> include/linux/memcontrol.h | 79 ++++++++++++++++++++++
>>> mm/memcontrol.c | 185 ++
>>> 2 files changed, 264 insertions(+)
>>>
>>> diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
>>> index 8d9489f..75b247e 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 +400,11 @@ struct sock;
>>>  #ifdef CONFIG_MEMCG_KMEM
>>>  void sock_update_memcg(struct sock *sk);
>>>  void sock_release_memcg(struct sock *sk);
>>> +
>>> +#define memcg_kmem_on 1
>>> +bool __memcg_kmem_new_page(gfp_t gfp, void *handle, int order);
>>> +void __memcg_kmem_commit_page(struct page *page, void *handle, int order);
>>> +void __memcg_kmem_free_page(struct page *page, int order);
>>>  #else
>>>  static inline void sock_update_memcg(struct sock *sk)
>>>  {
>>> @@ -406,6 +412,79 @@ static inline void sock_update_memcg(struct sock *sk)
>>>  static inline void sock_release_memcg(struct sock *sk)
>>>  {
>>>  }
>>> +
>>> +#define memcg_kmem_on 0
>>> +static inline bool
>>> +__memcg_kmem_new_page(gfp_t gfp, void *handle, int order)
>>> +{
>>> + return false;
>>> +}
>>> +
>>> +static inline void  __memcg_kmem_free_page(struct page *page, int order)
>>> +{
>>> +}
>>> +
>>> +static inline void
>>> +__memcg_kmem_commit_page(struct page *page, struct mem_cgroup *handle, int order)
>>> +{
>>> +}
>>>  #endif /* CONFIG_MEMCG_KMEM */
>>> +
>>> +/**
>>> + * memcg_kmem_new_page: verify if a new kmem allocation is allowed.
>>> + * @gfp: the gfp allocation flags.
>>> + * @handle: 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_new_page(gfp_t gfp, void *handle, int order)
>>> +{
>>> + if (!memcg_kmem_on)
>>> + return true;
>>> + if (!(gfp & __GFP_KMEMCG) || (gfp & __GFP_NOFAIL))
>>> + return true;
>>> + if (in_interrupt() || (!current->mm) || (current->flags & PF_KTHREAD))
>>> + return true;
>>> + return __memcg_kmem_new_page(gfp, handle, order);
>>> +}
>>> +
>>> +/**
>>> + * memcg_kmem_free_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_free_page(struct page *page, int order)
>>> +{
>>> + if (memcg_kmem_on)
>>> + __memcg_kmem_free_page(page, order);
>>> +}
>>> +
>>> +/**
>>> + * memcg_kmem_commit_page: embeds correct memcg in a page
>>> + * @handle: a pointer to the memcg this was charged against.
>>> + * @page: pointer to struct page recently allocated
>>> + * @handle: the memcg structure we charged against
>>> + * @order: allocation order.
>>> + *
>>> + * Needs to be called after memcg_kmem_new_page, 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 @handle to the
>>> + * corresponding page_cgroup.
>>> + */
>>> +static __always_inline void
>>> +memcg_kmem_commit_page(struct page *page, struct mem_cgroup *handle, int order)
>>> +{
>>> + if (memcg_kmem_on)
>>> + __memcg_kmem_commit_page(page, handle, order);
>>> +}
>>> #endif /* _LINUX_MEMCONTROL_H */
>>>
>>> diff --git a/mm/memcontrol.c b/mm/memcontrol.c

>>> index 54e93de..e9824c1 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
>>> @@ -434,6 +438,9 @@ struct mem_cgroup *mem_cgroup_from_css(struct
cgroup_subsys_state *s)
>>> #include <net/ip.h>
>>>
>>> static bool mem_cgroup_is_root(struct mem_cgroup *memcg);
>>> +static int memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t gfp, s64 delta);
>>> +static void memcg_uncharge_kmem(struct mem_cgroup *memcg, s64 delta);
>>> +
>>> void sock_update_memcg(struct sock *sk)
>>> {
>>> if (mem_cgroup_sockets_enabled) {
>>> @@ -488,6 +495,118 @@ struct cg_proto *tcp_proto_cgroup(struct mem_cgroup *memcg)
>>> }
>>> EXPORT_SYMBOL(tcp_proto_cgroup);
>>> #endif /* CONFIG_INET */
>>> +
>>> +static inline bool memcg_kmem_enabled(struct mem_cgroup *memcg)
>>> +{
>>> + return !mem_cgroup_disabled() && !mem_cgroup_is_root(memcg) &&
>>> + memcg->kmem_accounted;
>>> +}
>>> +
>>> +/*
>>> + * 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
>>> + *
>>> + * Returning true means the allocation is possible.
>>> + */

```
>>> +bool __memcg_kmem_new_page(gfp_t gfp, void *_handle, int order)
>>> +{
>>> + struct mem_cgroup *memcg;
>>> + struct mem_cgroup **handle = (struct mem_cgroup **)_handle;
>>> + bool ret = true;
>>> + size_t size;
>>> + struct task_struct *p;
>>> +
>>> + *handle = NULL;
>>> + rcu_read_lock();
>>> + p = rcu_dereference(current->mm->owner);
>>> + memcg = mem_cgroup_from_task(p);
>>> + if (!memcg_kmem_enabled(memcg))
>>> +  goto out;
>>> +
>>> + mem_cgroup_get(memcg);
>>> +
>>> + size = PAGE_SIZE << order;
>>> + ret = memcg_charge_kmem(memcg, gfp, size) == 0;
>>> + if (!ret) {
>>> +  mem_cgroup_put(memcg);
>>> +  goto out;
>>> + }
>>> +
>>> + *handle = memcg;
>>> +out:
>>> + rcu_read_unlock();
>>> + return ret;
>>> +}
>>> +EXPORT_SYMBOL(__memcg_kmem_new_page);
>>> +
>>> +void __memcg_kmem_commit_page(struct page *page, void *handle, int order)
>>> +{
>>> + struct page_cgroup *pc;
>>> + struct mem_cgroup *memcg = handle;
>>> +
>>> + if (!memcg)
>>> +  return;
>>> +
>>> + WARN_ON(mem_cgroup_is_root(memcg));
>>> + /* The page allocation must have failed. Revert */
>>> + if (!page) {
>>> +  size_t size = PAGE_SIZE << order;
>>> +
>>> +  memcg_uncharge_kmem(memcg, size);
>>> +  mem_cgroup_put(memcg);
>>> +  return;
>>
```

>>> +
>>> + pc = lookup_page_cgroup(page);
>>> + lock_page_cgroup(pc);
>>> + pc->mem_cgroup = memcg;
>>> + SetPageCgroupUsed(pc);
>>> + unlock_page_cgroup(pc);
>>
>> I have no problem with the code here. But, out of curiosity, why do we
>> need to lock the pc here and below in __memcg_kmem_free_page()?
>>
>> For the allocating side, I don't think that migration or reclaim will be
>> manipulating this page. But is there something else that we need the
>> locking for?
>>
>> For the freeing side, it seems that anyone calling
>> __memcg_kmem_free_page() is going to be freeing a previously accounted
>> page.
>>
>> I imagine that if we did not need the locking we would still need some
>> memory barriers to make sure that modifications to the PG_lru are
>> serialized wrt. to kmem modifying PageCgroupUsed here.
>>
> Unlocking should do that, no?

Yes, I agree that your existing locking should provide the necessary
barriers.

>> Perhaps we're just trying to take a conservative initial implementation
>> which is consistent with user visible pages.
>>
>
> The way I see it, is not about being conservative, but rather about my
> physical safety. It is quite easy and natural to assume that "all
> modifications to page cgroup are done under lock". So someone modifying
> this later will likely find out about this exception in a rather
> unpleasant way. They know where I live, and guns for hire are everywhere.
>
> Note that it is not unreasonable to believe that we can modify this
> later. This can be a way out, for example, for the memcg lifecycle problem.
>
> I agree with your analysis and we can ultimately remove it, but if we
> cannot pinpoint any performance problems to here, maybe consistency
> wins. Also, the locking operation itself is a bit expensive, but the
> biggest price is the actual contention. If we'll have nobody contending
> for the same page_cgroup, the problem - if exists - shouldn't be that
> bad. And if we ever have, the lock is needed.

Sounds reasonable. Another reason we might have to eventually revisit

this lock is the fact that lock_page_cgroup() is not generally irq_safe.
I assume that slab pages may be freed in softirq and would thus (in an
upcoming patch series) call __memcg_kmem_free_page. There are a few
factors that might make it safe to grab this lock here (and below in
__memcg_kmem_free_page) from hard/softirq context:
* the pc lock is a per page bit spinlock. So we only need to worry
  about interrupting a task which holds the same page's lock to avoid
  deadlock.
* for accounted kernel pages, I am not aware of other code beyond
  __memcg_kmem_charge_page and __memcg_kmem_free_page which grab pc
  lock. So we shouldn't find __memcg_kmem_free_page() called from a
  context which interrupted a holder of the page's pc lock.

>>> +}
>>> +
>>> +void __memcg_kmem_free_page(struct page *page, int order)
>>> +{
>>> + struct mem_cgroup *memcg;
>>> + size_t size;
>>> + struct page_cgroup *pc;
>>> +
>>> + if (mem_cgroup_disabled())
>>> + return;
>>> +
>>> + pc = lookup_page_cgroup(page);
>>> + lock_page_cgroup(pc);
>>> + memcg = pc->mem_cgroup;
>>> + pc->mem_cgroup = NULL;
>>> + if (!PageCgroupUsed(pc)) {
>>
>> When do we expect to find PageCgroupUsed() unset in this routine? Is
>> this just to handle the race of someone enabling kmem accounting after
>> allocating a page and then later freeing that page?
>>
>
> All the time we have a valid memcg. It is marked Used at charge time, so
> this is how we differentiate between a tracked page and a non-tracked
> page. Note that even though we explicit mark the freeing call sites with
> free_allocated_page, etc, not all pc->memcg will be valid. There are
> unlimited memcgs, bypassed charges, GFP_NOFAIL allocations, etc.

Understood. Thanks.

Subject: Re: [PATCH v2 06/11] memcg: kmem controller infrastructure
Posted by Glauber Costa on Thu, 23 Aug 2012 07:51:31 GMT
View Forum Message <> Reply to Message

>>> Perhaps we're just trying to take a conservative initial implementation
>>> which is consistent with user visible pages.
>>>
>>
>> The way I see it, is not about being conservative, but rather about my
>> physical safety. It is quite easy and natural to assume that "all
>> modifications to page cgroup are done under lock". So someone modifying
>> this later will likely find out about this exception in a rather
>> unpleasant way. They know where I live, and guns for hire are everywhere.
>>
>> Note that it is not unreasonable to believe that we can modify this
>> later. This can be a way out, for example, for the memcg lifecycle problem.
>>
>> I agree with your analysis and we can ultimately remove it, but if we
>> cannot pinpoint any performance problems to here, maybe consistency
>> wins. Also, the locking operation itself is a bit expensive, but the
>> biggest price is the actual contention. If we'll have nobody contending
>> for the same page_cgroup, the problem - if exists - shouldn't be that
>> bad. And if we ever have, the lock is needed.
>
> Sounds reasonable. Another reason we might have to eventually revisit
> this lock is the fact that lock_page_cgroup() is not generally irq_safe.
> I assume that slab pages may be freed in softirq and would thus (in an
> upcoming patch series) call __memcg_kmem_free_page. There are a few
> factors that might make it safe to grab this lock here (and below in
> __memcg_kmem_free_page) from hard/softirq context:
> * the pc lock is a per page bit spinlock. So we only need to worry
>   about interrupting a task which holds the same page's lock to avoid
>   deadlock.
> * for accounted kernel pages, I am not aware of other code beyond
>   __memcg_kmem_charge_page and __memcg_kmem_free_page which grab pc
>   lock. So we shouldn't find __memcg_kmem_free_page() called from a
>   context which interrupted a holder of the page's pc lock.
>

All very right.

Subject: Re: [PATCH v2 09/11] memcg: propagate kmem limiting information to children
Posted by Glauber Costa on Thu, 23 Aug 2012 07:55:02 GMT
View Forum Message <> Reply to Message

On 08/23/2012 03:23 AM, Greg Thelen wrote:
> On Wed, Aug 22 2012, Glauber Costa wrote:
>
>>>>>
>>>>> I am fine with either, I just need a clear sign from you guys so I don't

>>>>> keep deimplementing and reimplementing this forever.
>>>>
>>>> I would be for make it simple now and go with additional features later
>>>> when there is a demand for them. Maybe we will have runtimg switch for
>>>> user memory accounting as well one day.
>>>>
>>>> But let's see what others think?
>>>
>>> In my use case memcg will either be disable or (enabled and kmem
>>> limiting enabled).
>>>
>>> I'm not sure I follow the discussion about history. Are we saying that
>>> once a kmem limit is set then kmem will be accounted/charged to memcg.
>>> Is this discussion about the static branches/etc that are autotuned the
>>> first time is enabled?
>>
>> No, the question is about when you unlimit a former kmem-limited memcg.
>>
>>> The first time its set there parts of the system
>>> will be adjusted in such a way that may impose a performance overhead
>>> (static branches, etc). Thereafter the performance cannot be regained
>>> without a reboot. This makes sense to me. Are we saying that
>>> kmem.limit_in_bytes will have three states?
>>
>> It is not about performance, about interface.
>>
>> Michal says that once a particular memcg was kmem-limited, it will keep
>> accounting pages, even if you make it unlimited. The limits won't be
>> enforced, for sure - there is no limit, but pages will still be accounted.
>>
>> This simplifies the code galore, but I worry about the interface: A
>> person looking at the current status of the files only, without
>> knowledge of past history, can't tell if allocations will be tracked or not.
>
> In the current patch set we've conflating enabling kmem accounting with
> the kmem limit value (RESOURCE_MAX=disabled, all_other_values=enabled).
>
> I see no problem with simpling the kernel code with the requirement that
> once a particular memcg enables kmem accounting that it cannot be
> disabled for that memcg.
>
> The only question is the user space interface. Two options spring to
> mind:
> a) Close to current code. Once kmem.limit_in_bytes is set to
>   non-RESOURCE_MAX, then kmem accounting is enabled and cannot be
>   disabled. Therefore the limit cannot be set to RESOURCE_MAX
>   thereafter. The largest value would be something like
>   RESOURCE_MAX-PAGE_SIZE. An admin wondering if kmem is enabled only

> has to cat kmem.limit_in_bytes - if it's less than RESOURCE_MAX, then
> kmem is enabled.
>

If we need to choose between them, I like this better than your (b).
At least it is all clear, and "fix" the history problem, since it is
possible to look up the status of the files and figure it out.

> b) Or, if we could introduce a separate sticky kmem.enabled file. Once
> set it could not be unset. Kmem accounting would only be enabled if
> kmem.enabled=1.
>
> I think (b) is clearer.
>
Depends on your definition of clearer. We had a knob for
kmem_independent in the beginning if you remember, and it was removed.
The main reason being knobs complicate minds, and we happen to have a
very natural signal for this. I believe the same reasoning applies here.

Subject: Re: [PATCH v2 09/11] memcg: propagate kmem limiting information to children
Posted by Greg Thelen on Fri, 24 Aug 2012 05:06:50 GMT
View Forum Message <> Reply to Message

On Thu, Aug 23 2012, Glauber Costa wrote:

> On 08/23/2012 03:23 AM, Greg Thelen wrote:
>> On Wed, Aug 22 2012, Glauber Costa wrote:
>>
>>>>>>
>>>>>> I am fine with either, I just need a clear sign from you guys so I don't
>>>>>> keep deimplementing and reimplementing this forever.
>>>>>
>>>>> I would be for make it simple now and go with additional features later
>>>>> when there is a demand for them. Maybe we will have runtimg switch for
>>>>> user memory accounting as well one day.
>>>>>
>>>>> But let's see what others think?
>>>>
>>>> In my use case memcg will either be disable or (enabled and kmem
>>>> limiting enabled).
>>>>
>>>> I'm not sure I follow the discussion about history. Are we saying that
>>>> once a kmem limit is set then kmem will be accounted/charged to memcg.
>>>> Is this discussion about the static branches/etc that are autotuned the
>>>> first time is enabled?
>>>

>>> No, the question is about when you unlimit a former kmem-limited memcg.
>>>
>>>> The first time its set there parts of the system
>>>> will be adjusted in such a way that may impose a performance overhead
>>>> (static branches, etc). Thereafter the performance cannot be regained
>>>> without a reboot. This makes sense to me. Are we saying that
>>>> kmem.limit_in_bytes will have three states?
>>>
>>> It is not about performance, about interface.
>>>
>>> Michal says that once a particular memcg was kmem-limited, it will keep
>>> accounting pages, even if you make it unlimited. The limits won't be
>>> enforced, for sure - there is no limit, but pages will still be accounted.
>>>
>>> This simplifies the code galore, but I worry about the interface: A
>>> person looking at the current status of the files only, without
>>> knowledge of past history, can't tell if allocations will be tracked or not.
>>
>> In the current patch set we've conflating enabling kmem accounting with
>> the kmem limit value (RESOURCE_MAX=disabled, all_other_values=enabled).
>>
>> I see no problem with simpling the kernel code with the requirement that
>> once a particular memcg enables kmem accounting that it cannot be
>> disabled for that memcg.
>>
>> The only question is the user space interface. Two options spring to
>> mind:
>> a) Close to current code. Once kmem.limit_in_bytes is set to
>>   non-RESOURCE_MAX, then kmem accounting is enabled and cannot be
>>   disabled. Therefore the limit cannot be set to RESOURCE_MAX
>>   thereafter. The largest value would be something like
>>   RESOURCE_MAX-PAGE_SIZE. An admin wondering if kmem is enabled only
>>   has to cat kmem.limit_in_bytes - if it's less than RESOURCE_MAX, then
>>   kmem is enabled.
>>
>
> If we need to choose between them, I like this better than your (b).
> At least it is all clear, and "fix" the history problem, since it is
> possible to look up the status of the files and figure it out.
>
>> b) Or, if we could introduce a separate sticky kmem.enabled file. Once
>>   set it could not be unset. Kmem accounting would only be enabled if
>>   kmem.enabled=1.
>>
>> I think (b) is clearer.
>>
> Depends on your definition of clearer. We had a knob for
> kmem_independent in the beginning if you remember, and it was removed.

> The main reason being knobs complicate minds, and we happen to have a
> very natural signal for this. I believe the same reasoning applies here.

Sounds good to me, so let's go with (a).

Subject: Re: [PATCH v2 09/11] memcg: propagate kmem limiting information to children
Posted by Glauber Costa on Fri, 24 Aug 2012 05:23:58 GMT
View Forum Message <> Reply to Message

On 08/24/2012 09:06 AM, Greg Thelen wrote:
> On Thu, Aug 23 2012, Glauber Costa wrote:
>
>> On 08/23/2012 03:23 AM, Greg Thelen wrote:
>>> On Wed, Aug 22 2012, Glauber Costa wrote:
>>>
>>>>>>>> I am fine with either, I just need a clear sign from you guys so I don't
>>>>>>>> keep deimplementing and reimplementing this forever.
>>>>>>
>>>>>> I would be for make it simple now and go with additional features later
>>>>>> when there is a demand for them. Maybe we will have runtimg switch for
>>>>>> user memory accounting as well one day.
>>>>>>
>>>>>> But let's see what others think?
>>>>>
>>>>> In my use case memcg will either be disable or (enabled and kmem
>>>>> limiting enabled).
>>>>>
>>>>> I'm not sure I follow the discussion about history. Are we saying that
>>>>> once a kmem limit is set then kmem will be accounted/charged to memcg.
>>>>> Is this discussion about the static branches/etc that are autotuned the
>>>>> first time is enabled?
>>>>
>>>> No, the question is about when you unlimit a former kmem-limited memcg.
>>>>
>>> The first time its set there parts of the system
>>> will be adjusted in such a way that may impose a performance overhead
>>> (static branches, etc). Thereafter the performance cannot be regained
>>> without a reboot. This makes sense to me. Are we saying that
>>> kmem.limit_in_bytes will have three states?
>>>
>> It is not about performance, about interface.
>>
>> Michal says that once a particular memcg was kmem-limited, it will keep
>> accounting pages, even if you make it unlimited. The limits won't be
>> enforced, for sure - there is no limit, but pages will still be accounted.

>>>>
>>>> This simplifies the code galore, but I worry about the interface: A
>>>> person looking at the current status of the files only, without
>>>> knowledge of past history, can't tell if allocations will be tracked or not.
>>>
>>> In the current patch set we've conflating enabling kmem accounting with
>>> the kmem limit value (RESOURCE_MAX=disabled, all_other_values=enabled).
>>>
>>> I see no problem with simpling the kernel code with the requirement that
>>> once a particular memcg enables kmem accounting that it cannot be
>>> disabled for that memcg.
>>>
>>> The only question is the user space interface. Two options spring to
>>> mind:
>>> a) Close to current code. Once kmem.limit_in_bytes is set to
>>>   non-RESOURCE_MAX, then kmem accounting is enabled and cannot be
>>>   disabled. Therefore the limit cannot be set to RESOURCE_MAX
>>>   thereafter. The largest value would be something like
>>>   RESOURCE_MAX-PAGE_SIZE. An admin wondering if kmem is enabled only
>>>   has to cat kmem.limit_in_bytes - if it's less than RESOURCE_MAX, then
>>>   kmem is enabled.
>>>
>>
>> If we need to choose between them, I like this better than your (b).
>> At least it is all clear, and "fix" the history problem, since it is
>> possible to look up the status of the files and figure it out.
>>
>>> b) Or, if we could introduce a separate sticky kmem.enabled file. Once
>>>   set it could not be unset. Kmem accounting would only be enabled if
>>>   kmem.enabled=1.
>>>
>>> I think (b) is clearer.
>>>
>> Depends on your definition of clearer. We had a knob for
>> kmem_independent in the beginning if you remember, and it was removed.
>> The main reason being knobs complicate minds, and we happen to have a
>> very natural signal for this. I believe the same reasoning applies here.
>
> Sounds good to me, so let's go with (a).
>
Michal, what do you think ?
