Subject: [PATCH v3 00/13] kmem controller for memcg.
Posted by Glauber Costa on Tue, 18 Sep 2012 14:03:57 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.

*v3:
 - Changed function names to match memcg's
 - avoid doing get/put in charge/uncharge path
 - revert back to keeping the account enabled after it is first activated

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/. Patches for that will be sent shortly, and will include the documentation for this.

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

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

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

A general explanation of what this is all about follows:

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

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

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

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

In fact, a simple:

while true; do mkdir x; cd x; done

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

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

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

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

Glauber Costa (11):
  memcg: change defines to an enum
  kmem accounting basic infrastructure
  Add a __GFP_KMEMCG flag
  memcg: kmem controller infrastructure
  mm: Allocate kernel pages to the right memcg
  res_counter: return amount of charges after res_counter_uncharge
  memcg: kmem accounting lifecycle management
  memcg: use static branches when code not in use
  memcg: allow a memcg with kmem charges to be destructed.
  execute the whole memcg freeing in rcu callback
  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.

 Documentation/cgroups/resource_counter.txt | 7 +-
 include/linux/gfp.h | 10 +-
 include/linux/memcontrol.h | 99 ++++++
 include/linux/res_counter.h | 12 +-
 include/linux/thread_info.h | 2 +
 kernel/fork.c | 4 +-
 kernel/res_counter.c | 20 +-
 mm/memcontrol.c | 519 ++++++++++++++++++++++++++----
 mm/page_alloc.c | 35 ++
 9 files changed, 628 insertions(+), 80 deletions(-)

--
1.7.11.4

Subject: [PATCH v3 01/13] memcg: Make it possible to use the stock for more than one page.
Posted by Glauber Costa on Tue, 18 Sep 2012 14:03:58 GMT
View Forum Message <> Reply to Message

From: Suleiman Souhlal <ssouhlal@FreeBSD.org>

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

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

[glommer@parallels.com: added a changelog]

Signed-off-by: Suleiman Souhlal <suleiman@google.com>
Signed-off-by: Glauber Costa <glommer@parallels.com>
Acked-by: David Rientjes <rientjes@google.com>
Acked-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
Acked-by: Michal Hocko <mhocko@suse.cz>
---
 mm/memcontrol.c | 28 ++++++++++++++++++----------
 1 file changed, 18 insertions(+), 10 deletions(-)

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 795e525..9d3bc72 100644
--- a/mm/memcontrol.c

```
+++ b/mm/memcontrol.c
@@ -2034,20 +2034,28 @@ 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);
@@ -2346,7 +2354,7 @@ 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 {
@@ -2371,7 +2379,7 @@ 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
--
1.7.11.4
```

Subject: [PATCH v3 03/13] memcg: change defines to an enum
Posted by Glauber Costa on Tue, 18 Sep 2012 14:04:00 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: 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 b12121b..d6ad138 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -385,9 +385,12 @@ 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)
@@ -3921,7 +3924,8 @@ 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);
@@ -3957,7 +3961,8 @@ 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;

@@ -4033,7 +4038,8 @@ 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);
@@ -4369,7 +4375,7 @@ 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;

@@ -4452,7 +4458,7 @@ 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;
```

```
@@ -4530,7 +4536,7 @@ 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);
@@ -4555,7 +4561,7 @@ 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);

--
1.7.11.4
```

Subject: [PATCH v3 05/13] Add a __GFP_KMEMCG flag
Posted by Glauber Costa on Tue, 18 Sep 2012 14:04:02 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: 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>
Acked-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
---
 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).
--
1.7.11.4
```

Subject: [PATCH v3 06/13] memcg: kmem controller infrastructure
Posted by Glauber Costa on Tue, 18 Sep 2012 14:04:03 GMT
View Forum Message <> Reply to Message

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

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

[v2: improved comments and standardized function names]
[v3: handle no longer opaque, functions not exported,
  even more comments]
[v4: reworked Used bit handling and surroundings for more clarity]

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 |  97 ++++++++++++++++++++++++++
 mm/memcontrol.c            | 177 +++++++++++++++++++++++++++++++++++++++++++++++
 2 files changed, 274 insertions(+)

diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
index 8d9489f..82ede9a 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 +400,17 @@ struct sock;
 #ifdef CONFIG_MEMCG_KMEM
 void sock_update_memcg(struct sock *sk);
 void sock_release_memcg(struct sock *sk);
+
+static inline bool memcg_kmem_enabled(void)
+{
+ return true;
+}
+
+extern bool __memcg_kmem_newpage_charge(gfp_t gfp, struct mem_cgroup **memcg,
+    int order);
+extern void __memcg_kmem_commit_charge(struct page *page,
+          struct mem_cgroup *memcg, int order);
+extern void __memcg_kmem_uncharge_page(struct page *page, int order);
 #else
 static inline void sock_update_memcg(struct sock *sk)
 {
@@ -406,6 +418,91 @@ static inline void sock_update_memcg(struct sock *sk)
 static inline void sock_release_memcg(struct sock *sk)
 {
 }
+
+static inline bool memcg_kmem_enabled(void)
+{
+ return false;
+}
+
```

```
+static inline bool
+__memcg_kmem_newpage_charge(gfp_t gfp, struct mem_cgroup **memcg, int order)
+{
+ return false;
+}
+
+static inline void  __memcg_kmem_uncharge_page(struct page *page, int order)
+{
+}
+
+static inline void
+__memcg_kmem_commit_charge(struct page *page, struct mem_cgroup *memcg, int order)
+{
+}
 #endif /* CONFIG_MEMCG_KMEM */
+
+/**
+ * memcg_kmem_newpage_charge: verify if a new kmem allocation is allowed.
+ * @gfp: the gfp allocation flags.
+ * @memcg: a pointer to the memcg this was charged against.
+ * @order: allocation order.
+ *
+ * returns true if the memcg where the current task belongs can hold this
+ * allocation.
+ *
+ * We return true automatically if this allocation is not to be accounted to
+ * any memcg.
+ */
+static __always_inline bool
+memcg_kmem_newpage_charge(gfp_t gfp, struct mem_cgroup **memcg, int order)
+{
+ if (!memcg_kmem_enabled())
+  return true;
+
+ /*
+  * __GFP_NOFAIL allocations will move on even if charging is not
+  * possible. Therefore we don't even try, and have this allocation
+  * unaccounted. We could in theory charge it with
+  * res_counter_charge_nofail, but we hope those allocations are rare,
+  * and won't be worth the trouble.
+  */
+ if (!(gfp & __GFP_KMEMCG) || (gfp & __GFP_NOFAIL))
+  return true;
+ if (in_interrupt() || (!current->mm) || (current->flags & PF_KTHREAD))
+  return true;
+ return __memcg_kmem_newpage_charge(gfp, memcg, order);
+}
+
```

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

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index f3fd354..0f36a01 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
@@ -426,6 +430,9 @@ 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, u64 size);
+static void memcg_uncharge_kmem(struct mem_cgroup *memcg, u64 size);
+
 void sock_update_memcg(struct sock *sk)
 {
  if (mem_cgroup_sockets_enabled) {
@@ -480,6 +487,110 @@ struct cg_proto *tcp_proto_cgroup(struct mem_cgroup *memcg)
 }
 EXPORT_SYMBOL(tcp_proto_cgroup);
 #endif /* CONFIG_INET */
+
+static inline bool memcg_can_account_kmem(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. We could also return the memcg directly and avoid the pointer
+ * passing, but a boolean return value gives better semantics considering
+ * the compiled-out case as well.
+ *
+ * Returning true means the allocation is possible.
+ */
+bool
+__memcg_kmem_newpage_charge(gfp_t gfp, struct mem_cgroup **_memcg, int order)
+{
+ struct mem_cgroup *memcg;
+ bool ret;
+ struct task_struct *p;
+
+ *_memcg = NULL;
+ rcu_read_lock();
+ p = rcu_dereference(current->mm->owner);
+ memcg = mem_cgroup_from_task(p);
```

```
+ rcu_read_unlock();
+
+ if (!memcg_can_account_kmem(memcg))
+  return true;
+
+ mem_cgroup_get(memcg);
+
+ ret = memcg_charge_kmem(memcg, gfp, PAGE_SIZE << order) == 0;
+ if (ret)
+  *_memcg = memcg;
+ else
+  mem_cgroup_put(memcg);
+
+ return ret;
+}
+
+void __memcg_kmem_commit_charge(struct page *page, struct mem_cgroup *memcg,
+         int order)
+{
+ struct page_cgroup *pc;
+
+ WARN_ON(mem_cgroup_is_root(memcg));
+
+ /* The page allocation failed. Revert */
+ if (!page) {
+  memcg_uncharge_kmem(memcg, PAGE_SIZE << order);
+  return;
+ }
+
+ pc = lookup_page_cgroup(page);
+ lock_page_cgroup(pc);
+ pc->mem_cgroup = memcg;
+ SetPageCgroupUsed(pc);
+ unlock_page_cgroup(pc);
+}
+
+void __memcg_kmem_uncharge_page(struct page *page, int order)
+{
+ struct mem_cgroup *memcg = NULL;
+ struct page_cgroup *pc;
+
+
+ pc = lookup_page_cgroup(page);
+ /*
+  * Fast unlocked return. Theoretically might have changed, have to
+  * check again after locking.
+  */
+ if (!PageCgroupUsed(pc))
```

```
+  return;
+
+ lock_page_cgroup(pc);
+ if (PageCgroupUsed(pc)) {
+  memcg = pc->mem_cgroup;
+  ClearPageCgroupUsed(pc);
+ }
+ unlock_page_cgroup(pc);
+
+ /*
+  * 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));
+ memcg_uncharge_kmem(memcg, PAGE_SIZE << order);
+ mem_cgroup_put(memcg);
+}
 #endif /* CONFIG_MEMCG_KMEM */

 #if defined(CONFIG_INET) && defined(CONFIG_MEMCG_KMEM)
@@ -5700,3 +5811,69 @@ 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, u64 size)
+{
+ 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, size / 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, size, &fail_res);
+  if (do_swap_account)
+   res_counter_charge_nofail(&memcg->memsw, size,
+         &fail_res);
+  ret = 0;
+ } else if (ret == -ENOMEM)
+  return ret;
+
+ if (nofail)
+  res_counter_charge_nofail(&memcg->kmem, size, &fail_res);
+ else
+  ret = res_counter_charge(&memcg->kmem, size, &fail_res);
+
+ if (ret) {
+  res_counter_uncharge(&memcg->res, size);
+  if (do_swap_account)
+   res_counter_uncharge(&memcg->memsw, size);
+ }
+
+ return ret;
+}
+
+void memcg_uncharge_kmem(struct mem_cgroup *memcg, u64 size)
+{
+ if (!memcg)
+  return;
+
+ res_counter_uncharge(&memcg->kmem, size);
+ res_counter_uncharge(&memcg->res, size);
+ if (do_swap_account)
+  res_counter_uncharge(&memcg->memsw, size);
+}
```

+#endif /* CONFIG_MEMCG_KMEM */
--
1.7.11.4

Subject: [PATCH v3 07/13] mm: Allocate kernel pages to the right memcg
Posted by Glauber Costa on Tue, 18 Sep 2012 14:04:04 GMT
View Forum Message <> Reply to Message

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

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

[v2: inverted test order to avoid a memcg_get leak,
  free_accounted_pages simplification]

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: Johannes Weiner <hannes@cmpxchg.org>
CC: Suleiman Souhlal <suleiman@google.com>
CC: Mel Gorman <mgorman@suse.de>
Acked-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
---
 include/linux/gfp.h | 3 +++
 mm/page_alloc.c | 35 +++++++++++++++++++++++++++++++++++
 2 files changed, 38 insertions(+)

diff --git a/include/linux/gfp.h b/include/linux/gfp.h
index 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 b0c5a52..897d8e2 100644
--- a/mm/page_alloc.c
+++ b/mm/page_alloc.c
@@ -2573,6 +2573,7 @@ __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;
+ struct mem_cgroup *memcg = NULL;

  gfp_mask &= gfp_allowed_mask;

@@ -2591,6 +2592,13 @@ __alloc_pages_nodemask(gfp_t gfp_mask, unsigned int order,
  if (unlikely(!zonelist->_zonerefs->zone))
   return NULL;

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

@@ -2624,6 +2632,8 @@ out:
  if (unlikely(!put_mems_allowed(cpuset_mems_cookie) && !page))
   goto retry_cpuset;

+ memcg_kmem_commit_charge(page, memcg, order);
+
  return page;
 }
 EXPORT_SYMBOL(__alloc_pages_nodemask);
@@ -2676,6 +2686,31 @@ void free_pages(unsigned long addr, unsigned int order)

 EXPORT_SYMBOL(free_pages);

+/*
+ * __free_accounted_pages and free_accounted_pages will free pages allocated
+ * with __GFP_KMEMCG.
+ *
+ * Those pages are accounted to a particular memcg, embedded in the
+ * corresponding page_cgroup. To avoid adding a hit in the allocator to search
+ * for that information only to find out that it is NULL for users who have no
```

```
+ * interest in that whatsoever, we provide these functions.
+ *
+ * The caller knows better which flags it relies on.
+ */
+void __free_accounted_pages(struct page *page, unsigned int order)
+{
+ memcg_kmem_uncharge_page(page, order);
+ __free_pages(page, order);
+}
+
+void free_accounted_pages(unsigned long addr, unsigned int order)
+{
+ if (addr != 0) {
+  VM_BUG_ON(!virt_addr_valid((void *)addr));
+  __free_accounted_pages(virt_to_page((void *)addr), order);
+ }
+}
+
 static void *make_alloc_exact(unsigned long addr, unsigned order, size_t size)
 {
  if (addr) {
--
1.7.11.4
```

Subject: [PATCH v3 09/13] memcg: kmem accounting lifecycle management
Posted by Glauber Costa on Tue, 18 Sep 2012 14:04:06 GMT
View Forum Message <> Reply to Message

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.

-> /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
downwards and make sure that we store the information about the parent

being limited in kmem_accounted.

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

Since kmem charges may outlive the cgroup existance, we need to be extra
careful to guarantee the memcg object will stay around for as long as
needed. Up to now, we were using a mem_cgroup_get()/put() pair in charge
and uncharge operations.

Although this guarantees that the object will be around until the last
call to unchage, this means an atomic update in every charge. We can do
better than that if we only issue get() in the first charge, and then
put() when the last charge finally goes away.

[v3: merged all lifecycle related patches in one]

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: 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 | 123 ++-----
 1 file changed, 112 insertions(+), 11 deletions(-)

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 0f36a01..720e4bb 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -287,7 +287,8 @@ 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;
@@ -340,6 +341,43 @@ struct mem_cgroup {
 #endif
 };

+enum {
+ KMEM_ACCOUNTED_ACTIVE = 0, /* accounted by this cgroup itself */

```
+ KMEM_ACCOUNTED_PARENT, /* one of its parents is active */
+ KMEM_ACCOUNTED_DEAD, /* dead memcg, pending kmem charges */
+};
+
+/* bits 0 and 1 */
+#define KMEM_ACCOUNTED_MASK 0x3
+
+#ifdef CONFIG_MEMCG_KMEM
+static bool memcg_kmem_set_active(struct mem_cgroup *memcg)
+{
+ return !test_and_set_bit(KMEM_ACCOUNTED_ACTIVE, &memcg->kmem_accounted);
+}
+
+static bool memcg_kmem_is_accounted(struct mem_cgroup *memcg)
+{
+ return test_bit(KMEM_ACCOUNTED_ACTIVE, &memcg->kmem_accounted);
+}
+
+static void memcg_kmem_set_active_parent(struct mem_cgroup *memcg)
+{
+ set_bit(KMEM_ACCOUNTED_PARENT, &memcg->kmem_accounted);
+}
+
+static void memcg_kmem_mark_dead(struct mem_cgroup *memcg)
+{
+ if (test_bit(KMEM_ACCOUNTED_ACTIVE, &memcg->kmem_accounted))
+  set_bit(KMEM_ACCOUNTED_DEAD, &memcg->kmem_accounted);
+}
+
+static bool memcg_kmem_dead(struct mem_cgroup *memcg)
+{
+ return test_and_clear_bit(KMEM_ACCOUNTED_DEAD, &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
@@ -491,7 +529,7 @@ EXPORT_SYMBOL(tcp_proto_cgroup);
 static inline bool memcg_can_account_kmem(struct mem_cgroup *memcg)
 {
  return !mem_cgroup_disabled() && !mem_cgroup_is_root(memcg) &&
-  memcg->kmem_accounted;
+   (memcg->kmem_accounted & (KMEM_ACCOUNTED_MASK));
 }

 /*
@@ -524,13 +562,9 @@ __memcg_kmem_newpage_charge(gfp_t gfp, struct mem_cgroup
```

```
**_memcg, int order)
  if (!memcg_can_account_kmem(memcg))
   return true;

- mem_cgroup_get(memcg);
-
  ret = memcg_charge_kmem(memcg, gfp, PAGE_SIZE << order) == 0;
  if (ret)
   *_memcg = memcg;
- else
-  mem_cgroup_put(memcg);

  return ret;
 }
@@ -589,7 +623,6 @@ void __memcg_kmem_uncharge_page(struct page *page, int order)

  WARN_ON(mem_cgroup_is_root(memcg));
  memcg_uncharge_kmem(memcg, PAGE_SIZE << order);
- mem_cgroup_put(memcg);
 }
 #endif /* CONFIG_MEMCG_KMEM */

@@ -4077,6 +4110,40 @@ 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
+ struct mem_cgroup *iter;
+
+ /*
+  * When we are doing hierarchical accounting, with an hierarchy like
+  * A/B/C, we need to start accounting kernel memory all the way up to C
+  * in case A start being accounted.
+  *
+  * So when we the cgroup first gets to be unlimited, we walk all the
+  * children of the current memcg and enable kmem accounting for them.
+  * Note that a separate bit is used there to indicate that the
+  * accounting happens due to the parent being accounted.
+  *
+  * note that memcg_kmem_set_active is a test-and-set routine, so we only
+  * arrive here once (since we never disable it)
+  */
+ mutex_lock(&set_limit_mutex);
+ if ((val != RESOURCE_MAX) && memcg_kmem_set_active(memcg)) {
```

```
+
+  mem_cgroup_get(memcg);
+
+  for_each_mem_cgroup_tree(iter, memcg) {
+   if (iter == memcg)
+    continue;
+   memcg_kmem_set_active_parent(iter);
+  }
+ }
+ mutex_unlock(&set_limit_mutex);
+#endif
+}
+
 /*
  * The user of this function is...
  * RES_LIMIT.
@@ -4115,9 +4182,7 @@ static int mem_cgroup_write(struct cgroup *cont, struct cftype *cft,
    if (ret)
     break;

-   /* For simplicity, we won't allow this to be disabled */
-   if (!memcg->kmem_accounted && val != RESOURCE_MAX)
-    memcg->kmem_accounted = true;
+   memcg_update_kmem_limit(memcg, val);
   } else
    return -EINVAL;
   break;
@@ -4791,6 +4856,20 @@ static int memcg_init_kmem(struct mem_cgroup *memcg, struct
cgroup_subsys *ss)
 static void kmem_cgroup_destroy(struct mem_cgroup *memcg)
 {
  mem_cgroup_sockets_destroy(memcg);
+
+ memcg_kmem_mark_dead(memcg);
+
+ if (res_counter_read_u64(&memcg->kmem, RES_USAGE) != 0)
+  return;
+
+ /*
+  * Charges already down to 0, undo mem_cgroup_get() done in the charge
+  * path here, being careful not to race with memcg_uncharge_kmem: it is
+  * possible that the charges went down to 0 between mark_dead and the
+  * res_counter read, so in that case, we don't need the put
+  */
+ if (memcg_kmem_dead(memcg))
+  mem_cgroup_put(memcg);
 }
 #else
```

```
 static int memcg_init_kmem(struct mem_cgroup *memcg, struct cgroup_subsys *ss)
@@ -5148,6 +5227,8 @@ mem_cgroup_create(struct cgroup *cont)
  }

  if (parent && parent->use_hierarchy) {
+  struct mem_cgroup __maybe_unused *p;
+
   res_counter_init(&memcg->res, &parent->res);
   res_counter_init(&memcg->memsw, &parent->memsw);
   res_counter_init(&memcg->kmem, &parent->kmem);
@@ -5158,6 +5239,20 @@ mem_cgroup_create(struct cgroup *cont)
    * mem_cgroup(see mem_cgroup_put).
    */
   mem_cgroup_get(parent);
+#ifdef CONFIG_MEMCG_KMEM
+  /*
+   * In case a parent is already limited when we create this, we
+   * need him to propagate it now so we become limited as well.
+   */
+  mutex_lock(&set_limit_mutex);
+  for (p = parent; p != NULL; p = parent_mem_cgroup(p)) {
+   if (memcg_kmem_is_accounted(p)) {
+    memcg_kmem_set_active_parent(memcg);
+    break;
+   }
+  }
+  mutex_unlock(&set_limit_mutex);
+#endif
  } else {
   res_counter_init(&memcg->res, NULL);
   res_counter_init(&memcg->memsw, NULL);
@@ -5871,9 +5966,15 @@ void memcg_uncharge_kmem(struct mem_cgroup *memcg, u64
size)
  if (!memcg)
   return;

- res_counter_uncharge(&memcg->kmem, size);
  res_counter_uncharge(&memcg->res, size);
  if (do_swap_account)
   res_counter_uncharge(&memcg->memsw, size);
+
+ /* Not down to 0 */
+ if (res_counter_uncharge(&memcg->kmem, size))
+  return;
+
+ if (memcg_kmem_dead(memcg))
+  mem_cgroup_put(memcg);
 }
```

```
 #endif /* CONFIG_MEMCG_KMEM */
--
1.7.11.4
```

Subject: [PATCH v3 10/13] memcg: use static branches when code not in use
Posted by Glauber Costa on Tue, 18 Sep 2012 14:04:07 GMT
View Forum Message <> Reply to Message

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

Because the _ACTIVATED bit on kmem_accounted is only set once, 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.

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

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: 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 | 4 +++-
 mm/memcontrol.c | 26 ++++++++++++++++++++++++--
 2 files changed, 27 insertions(+), 3 deletions(-)

diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
index 82ede9a..4ec9fd5 100644
--- a/include/linux/memcontrol.h
+++ b/include/linux/memcontrol.h
@@ -22,6 +22,7 @@
 #include <linux/cgroup.h>
 #include <linux/vm_event_item.h>
 #include <linux/hardirq.h>
+#include <linux/jump_label.h>

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

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

 extern bool __memcg_kmem_newpage_charge(gfp_t gfp, struct mem_cgroup **memcg,
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 720e4bb..aada601 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -467,6 +467,8 @@ 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;
+
 static bool mem_cgroup_is_root(struct mem_cgroup *memcg);
 static int memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t gfp, u64 size);
 static void memcg_uncharge_kmem(struct mem_cgroup *memcg, u64 size);
@@ -624,6 +626,16 @@ void __memcg_kmem_uncharge_page(struct page *page, int order)
  WARN_ON(mem_cgroup_is_root(memcg));
  memcg_uncharge_kmem(memcg, PAGE_SIZE << order);
 }
+
+static void disarm_kmem_keys(struct mem_cgroup *memcg)
+{
+ if (memcg_kmem_is_accounted(memcg))
+  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)
@@ -639,6 +651,12 @@ 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 *
@@ -4131,7 +4149,11 @@ static void memcg_update_kmem_limit(struct mem_cgroup *memcg,
u64 val)
   */
  mutex_lock(&set_limit_mutex);
  if ((val != RESOURCE_MAX) && memcg_kmem_set_active(memcg)) {
-
+  /*
+   * Once the static branch is enabled it will only be
+   * disabled when the last reference to memcg is gone.
+   */
+  static_key_slow_inc(&memcg_kmem_enabled_key);
   mem_cgroup_get(memcg);

   for_each_mem_cgroup_tree(iter, memcg) {
@@ -5066,7 +5088,7 @@ 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
--
1.7.11.4
```

Subject: [PATCH v3 11/13] memcg: allow a memcg with kmem charges to be destructed.

Posted by Glauber Costa on Tue, 18 Sep 2012 14:04:08 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 aada601..b05ecac 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -631,6 +631,11 @@ static void disarm_kmem_keys(struct mem_cgroup *memcg)
 {
  if (memcg_kmem_is_accounted(memcg))
   static_key_slow_dec(&memcg_kmem_enabled_key);
+ /*
+  * This check can't live in kmem destruction function,
+  * since the charges will outlive the cgroup
+  */
+ WARN_ON(res_counter_read_u64(&memcg->kmem, RES_USAGE) != 0);
 }
 #else
 static void disarm_kmem_keys(struct mem_cgroup *memcg)
@@ -3933,6 +3938,7 @@ 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);

@@ -3966,8 +3972,17 @@ 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;
--
1.7.11.4
```

Subject: [PATCH v3 12/13] execute the whole memcg freeing in rcu callback
Posted by Glauber Costa on Tue, 18 Sep 2012 14:04:09 GMT
View Forum Message <> Reply to Message

A lot of the initialization we do in mem_cgroup_create() is done with softirqs
enabled. This include grabbing a css id, which holds &ss->id_lock->rlock, and
the per-zone trees, which holds rtpz->lock->rlock. All of those signal to the
lockdep mechanism that those locks can be used in SOFTIRQ-ON-W context. This
means that the freeing of memcg structure must happen in a compatible context,
otherwise we'll get a deadlock.

The reference counting mechanism we use allows the memcg structure to be freed
later and outlive the actual memcg destruction from the filesystem. However, we
have little, if any, means to guarantee in which context the last memcg_put
will happen. The best we can do is test it and try to make sure no invalid
context releases are happening. But as we add more code to memcg, the possible
interactions grow in number and expose more ways to get context conflicts.

We already moved a part of the freeing to a worker thread to be context-safe
for the static branches disabling. I see no reason not to do it for the whole
freeing action. I consider this to be the safe choice.

Signed-off-by: Glauber Costa <glommer@parallels.com>
Tested-by: Greg Thelen <gthelen@google.com>
CC: KAMEZAWA Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Michal Hocko <mhocko@suse.cz>
CC: Johannes Weiner <hannes@cmpxchg.org>

```
---
 mm/memcontrol.c | 66 ++++++++++++++++++++++++++++++++++---------------------------------
 1 file changed, 34 insertions(+), 32 deletions(-)

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index b05ecac..74654f0 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -5082,16 +5082,29 @@ out_free:
 }

 /*
```

```
- * Helpers for freeing a kmalloc()ed/vzalloc()ed mem_cgroup by RCU,
- * but in process context.  The work_freeing structure is overlaid
- * on the rcu_freeing structure, which itself is overlaid on memsw.
+ * At destroying mem_cgroup, references from swap_cgroup can remain.
+ * (scanning all at force_empty is too costly...)
+ *
+ * Instead of clearing all references at force_empty, we remember
+ * the number of reference from swap_cgroup and free mem_cgroup when
+ * it goes down to 0.
+ *
+ * Removal of cgroup itself succeeds regardless of refs from swap.
  */
-static void free_work(struct work_struct *work)
+
+static void __mem_cgroup_free(struct mem_cgroup *memcg)
 {
- struct mem_cgroup *memcg;
+ int node;
  int size = sizeof(struct mem_cgroup);

- memcg = container_of(work, struct mem_cgroup, work_freeing);
+ mem_cgroup_remove_from_trees(memcg);
+ free_css_id(&mem_cgroup_subsys, &memcg->css);
+
+ for_each_node(node)
+  free_mem_cgroup_per_zone_info(memcg, node);
+
+ free_percpu(memcg->stat);
+
  /*
   * We need to make sure that (at least for now), the jump label
   * destruction code runs outside of the cgroup lock. This is because
@@ -5110,38 +5123,27 @@ static void free_work(struct work_struct *work)
   vfree(memcg);
 }

-static void free_rcu(struct rcu_head *rcu_head)
-{
- struct mem_cgroup *memcg;
-
- memcg = container_of(rcu_head, struct mem_cgroup, rcu_freeing);
- INIT_WORK(&memcg->work_freeing, free_work);
- schedule_work(&memcg->work_freeing);
-}

 /*
- * At destroying mem_cgroup, references from swap_cgroup can remain.
- * (scanning all at force_empty is too costly...)
```

```
- *
- * Instead of clearing all references at force_empty, we remember
- * the number of reference from swap_cgroup and free mem_cgroup when
- * it goes down to 0.
- *
- * Removal of cgroup itself succeeds regardless of refs from swap.
+ * Helpers for freeing a kmalloc()ed/vzalloc()ed mem_cgroup by RCU,
+ * but in process context.  The work_freeing structure is overlaid
+ * on the rcu_freeing structure, which itself is overlaid on memsw.
  */
-
-static void __mem_cgroup_free(struct mem_cgroup *memcg)
+static void free_work(struct work_struct *work)
 {
- int node;
+ struct mem_cgroup *memcg;

- mem_cgroup_remove_from_trees(memcg);
- free_css_id(&mem_cgroup_subsys, &memcg->css);
+ memcg = container_of(work, struct mem_cgroup, work_freeing);
+ __mem_cgroup_free(memcg);
+}

- for_each_node(node)
-  free_mem_cgroup_per_zone_info(memcg, node);
+static void free_rcu(struct rcu_head *rcu_head)
+{
+ struct mem_cgroup *memcg;

- free_percpu(memcg->stat);
- call_rcu(&memcg->rcu_freeing, free_rcu);
+ memcg = container_of(rcu_head, struct mem_cgroup, rcu_freeing);
+ INIT_WORK(&memcg->work_freeing, free_work);
+ schedule_work(&memcg->work_freeing);
 }

 static void mem_cgroup_get(struct mem_cgroup *memcg)
@@ -5153,7 +5155,7 @@ static void __mem_cgroup_put(struct mem_cgroup *memcg, int count)
 {
  if (atomic_sub_and_test(count, &memcg->refcnt)) {
   struct mem_cgroup *parent = parent_mem_cgroup(memcg);
-  __mem_cgroup_free(memcg);
+  call_rcu(&memcg->rcu_freeing, free_rcu);
   if (parent)
    mem_cgroup_put(parent);
  }
--
1.7.11.4
```

Subject: [PATCH v3 13/13] protect architectures where THREAD_SIZE >= PAGE_SIZE against fork bombs
Posted by Glauber Costa on Tue, 18 Sep 2012 14:04:10 GMT
View Forum Message <> Reply to Message

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

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

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

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

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

Signed-off-by: Glauber Costa <glommer@parallels.com>
Acked-by: Frederic Weisbecker <fweisbec@redhat.com>
Acked-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
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/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 0ff2bf7..897e89c 100644
--- a/kernel/fork.c
+++ b/kernel/fork.c
@@ -146,7 +146,7 @@ void __weak arch_release_thread_info(struct thread_info *ti)
 static struct thread_info *alloc_thread_info_node(struct task_struct *tsk,
         int node)
 {
- struct page *page = alloc_pages_node(node, THREADINFO_GFP,
+ struct page *page = alloc_pages_node(node, THREADINFO_GFP_ACCOUNTED,
           THREAD_SIZE_ORDER);

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

 static inline void free_thread_info(struct thread_info *ti)
 {
- free_pages((unsigned long)ti, THREAD_SIZE_ORDER);
+ free_accounted_pages((unsigned long)ti, THREAD_SIZE_ORDER);
 }
 # else
 static struct kmem_cache *thread_info_cache;
--
1.7.11.4
```

Subject: Re: [PATCH v3 05/13] Add a __GFP_KMEMCG flag
Posted by Rik van Riel on Tue, 18 Sep 2012 14:15:18 GMT
View Forum Message <> Reply to Message

On 09/18/2012 10:04 AM, 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: 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>
> Acked-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>

Acked-by: Rik van Riel <riel@redhat.com>

--
All rights reversed

Subject: Re: [PATCH v3 05/13] Add a __GFP_KMEMCG flag
Posted by Christoph Lameter on Tue, 18 Sep 2012 15:06:22 GMT
View Forum Message <> Reply to Message

On Tue, 18 Sep 2012, Glauber Costa wrote:

> +++ 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

Could you leave __GFP_MEMCG a simple definition and then define GFP_MEMCG
to be zer0 if !MEMCG_KMEM? I think that would be cleaner and the
__GFP_KMEMCHECK another case that would be good to fix up.

Subject: Re: [PATCH v3 05/13] Add a __GFP_KMEMCG flag
Posted by Glauber Costa on Wed, 19 Sep 2012 07:39:33 GMT
View Forum Message <> Reply to Message

On 09/18/2012 07:06 PM, Christoph Lameter wrote:
> On Tue, 18 Sep 2012, Glauber Costa wrote:
>
>> +++ 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
>>
> Could you leave __GFP_MEMCG a simple definition and then define GFP_MEMCG

> to be zer0 if !MEMCG_KMEM? I think that would be cleaner and the
> __GFP_KMEMCHECK another case that would be good to fix up.
>
>
>
I can, but what does this buy us?
Also, in any case, this can be done incrementally, and for the other
flag as well, as you describe.

Subject: Re: [PATCH v3 05/13] Add a __GFP_KMEMCG flag
Posted by Christoph Lameter on Wed, 19 Sep 2012 14:07:33 GMT
View Forum Message <> Reply to Message

On Wed, 19 Sep 2012, Glauber Costa wrote:

> On 09/18/2012 07:06 PM, Christoph Lameter wrote:
> > On Tue, 18 Sep 2012, Glauber Costa wrote:
> >
> >> +++ 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
> >
> > Could you leave __GFP_MEMCG a simple definition and then define GFP_MEMCG
> > to be zer0 if !MEMCG_KMEM? I think that would be cleaner and the
> > __GFP_KMEMCHECK another case that would be good to fix up.
> >
> >
> >
> I can, but what does this buy us?

All the numeric values should be defined with __ unconditionally so that
they can be used in future context. Note the comment above the __GFP_XX
which says "Do not use this directly".

> Also, in any case, this can be done incrementally, and for the other
> flag as well, as you describe.

There is only one other flag that does not follow the scheme. I'd
appreciate it if you could submit a patch to fix up the __GFP_NOTRACK
conditional there.

There is no need to do this incrementally. Do it the right way
immediately.

Subject: Re: [PATCH v3 06/13] memcg: kmem controller infrastructure
Posted by JoonSoo Kim on Thu, 20 Sep 2012 16:05:47 GMT
View Forum Message <> Reply to Message

Hi, Glauber.

2012/9/18 Glauber Costa <glommer@parallels.com>:
> +/*
> + * We need to verify if the allocation against current->mm->owner's memcg is
> + * possible for the given order. But the page is not allocated yet, so we'll
> + * need a further commit step to do the final arrangements.
> + *
> + * It is possible for the task to switch cgroups in this mean time, so at
> + * commit time, we can't rely on task conversion any longer. We'll then use
> + * the handle argument to return to the caller which cgroup we should commit
> + * against. We could also return the memcg directly and avoid the pointer
> + * passing, but a boolean return value gives better semantics considering
> + * the compiled-out case as well.
> + *
> + * Returning true means the allocation is possible.
> + */
> +bool
> +__memcg_kmem_newpage_charge(gfp_t gfp, struct mem_cgroup **_memcg, int order)
> +{
> +	struct mem_cgroup *memcg;
> +	bool ret;
> +	struct task_struct *p;
> +
> +	*_memcg = NULL;
> +	rcu_read_lock();
> +	p = rcu_dereference(current->mm->owner);
> +	memcg = mem_cgroup_from_task(p);
> +	rcu_read_unlock();
> +
> +	if (!memcg_can_account_kmem(memcg))
> +		return true;
> +
> +	mem_cgroup_get(memcg);
> +
> +	ret = memcg_charge_kmem(memcg, gfp, PAGE_SIZE << order) == 0;
> +	if (ret)
> +		*_memcg = memcg;
> +	else

> +		mem_cgroup_put(memcg);
> +
> +	return ret;
> +}

"*_memcg = memcg" should be executed when "memcg_charge_kmem" is success.
"memcg_charge_kmem" return 0 if success in charging.
Therefore, I think this code is wrong.
If I am right, it is a serious bug that affect behavior of all the patchset.

> +void __memcg_kmem_commit_charge(struct page *page, struct mem_cgroup *memcg,
> +				int order)
> +{
> +	struct page_cgroup *pc;
> +
> +	WARN_ON(mem_cgroup_is_root(memcg));
> +
> +	/* The page allocation failed. Revert */
> +	if (!page) {
> +		memcg_uncharge_kmem(memcg, PAGE_SIZE << order);
> +		return;
> +	}

In case of "!page ", mem_cgroup_put(memcg) is needed,
because we already call "mem_cgroup_get(memcg)" in
__memcg_kmem_newpage_charge().
I know that mem_cgroup_put()/get() will be removed in later patch, but
it is important that every patch works fine.

Thanks.

Subject: Re: [PATCH v3 06/13] memcg: kmem controller infrastructure
Posted by Glauber Costa on Fri, 21 Sep 2012 08:41:58 GMT
View Forum Message <> Reply to Message

On 09/20/2012 08:05 PM, JoonSoo Kim wrote:
> Hi, Glauber.
>
> 2012/9/18 Glauber Costa <glommer@parallels.com>:
>> +/*
>> + * We need to verify if the allocation against current->mm->owner's memcg is
>> + * possible for the given order. But the page is not allocated yet, so we'll
>> + * need a further commit step to do the final arrangements.
>> + *
>> + * It is possible for the task to switch cgroups in this mean time, so at
>> + * commit time, we can't rely on task conversion any longer. We'll then use
>> + * the handle argument to return to the caller which cgroup we should commit

```
>> + * against. We could also return the memcg directly and avoid the pointer
>> + * passing, but a boolean return value gives better semantics considering
>> + * the compiled-out case as well.
>> + *
>> + * Returning true means the allocation is possible.
>> + */
>> +bool
>> +__memcg_kmem_newpage_charge(gfp_t gfp, struct mem_cgroup **_memcg, int order)
>> +{
>> +	struct mem_cgroup *memcg;
>> +	bool ret;
>> +	struct task_struct *p;
>> +
>> +	*_memcg = NULL;
>> +	rcu_read_lock();
>> +	p = rcu_dereference(current->mm->owner);
>> +	memcg = mem_cgroup_from_task(p);
>> +	rcu_read_unlock();
>> +
>> +	if (!memcg_can_account_kmem(memcg))
>> +		return true;
>> +
>> +	mem_cgroup_get(memcg);
>> +
>> +	ret = memcg_charge_kmem(memcg, gfp, PAGE_SIZE << order) == 0;
>> +	if (ret)
>> +		*_memcg = memcg;
>> +	else
>> +		mem_cgroup_put(memcg);
>> +
>> +	return ret;
>> +}
>
> "*_memcg = memcg" should be executed when "memcg_charge_kmem" is success.
> "memcg_charge_kmem" return 0 if success in charging.
> Therefore, I think this code is wrong.
> If I am right, it is a serious bug that affect behavior of all the patchset.

Which is precisely what it does. ret is a boolean, that will be true
when charge succeeded (== 0 test)

>
>> +void __memcg_kmem_commit_charge(struct page *page, struct mem_cgroup *memcg,
>> +				int order)
>> +{
>> +	struct page_cgroup *pc;
>> +
>> +	WARN_ON(mem_cgroup_is_root(memcg));
```

>> +
>> + /* The page allocation failed. Revert */
>> + if (!page) {
>> + memcg_uncharge_kmem(memcg, PAGE_SIZE << order);
>> + return;
>> + }
>
> In case of "!page ", mem_cgroup_put(memcg) is needed,
> because we already call "mem_cgroup_get(memcg)" in
> __memcg_kmem_newpage_charge().
> I know that mem_cgroup_put()/get() will be removed in later patch, but
> it is important that every patch works fine.

Okay, I'll add the put here. It is indeed missing.

Subject: Re: [PATCH v3 06/13] memcg: kmem controller infrastructure
Posted by JoonSoo Kim on Fri, 21 Sep 2012 09:14:10 GMT
View Forum Message <> Reply to Message

>> "*_memcg = memcg" should be executed when "memcg_charge_kmem" is success.
>> "memcg_charge_kmem" return 0 if success in charging.
>> Therefore, I think this code is wrong.
>> If I am right, it is a serious bug that affect behavior of all the patchset.
>
> Which is precisely what it does. ret is a boolean, that will be true
> when charge succeeded (== 0 test)

Ahh...Okay! I didn't see (== 0 test)

Subject: Re: [PATCH v3 12/13] execute the whole memcg freeing in rcu callback
Posted by Tejun Heo on Fri, 21 Sep 2012 17:23:55 GMT
View Forum Message <> Reply to Message

Hello, Glauber.

On Tue, Sep 18, 2012 at 06:04:09PM +0400, Glauber Costa wrote:
> A lot of the initialization we do in mem_cgroup_create() is done with softirqs
> enabled. This include grabbing a css id, which holds &ss->id_lock->rlock, and
> the per-zone trees, which holds rtpz->lock->rlock. All of those signal to the
> lockdep mechanism that those locks can be used in SOFTIRQ-ON-W context. This
> means that the freeing of memcg structure must happen in a compatible context,
> otherwise we'll get a deadlock.

Lockdep requires lock to be softirq or irq safe iff the lock is
actually acquired from the said context. Merely using a lock with bh

/ irq disabled doesn't signal that to lockdep; otherwise, we'll end up
with enormous number of spurious warnings.

> The reference counting mechanism we use allows the memcg structure to be freed
> later and outlive the actual memcg destruction from the filesystem. However, we
> have little, if any, means to guarantee in which context the last memcg_put
> will happen. The best we can do is test it and try to make sure no invalid
> context releases are happening. But as we add more code to memcg, the possible
> interactions grow in number and expose more ways to get context conflicts.
>
> We already moved a part of the freeing to a worker thread to be context-safe
> for the static branches disabling. I see no reason not to do it for the whole
> freeing action. I consider this to be the safe choice.

And the above description too makes me scratch my head quite a bit. I
can see what the patch is doing but can't understand the why.

* Why was it punting the freeing to workqueue anyway? ISTR something
  about static_keys but my memory fails. What changed? Why don't we
  need it anymore?

* As for locking context, the above description seems a bit misleading
  to me. Synchronization constructs involved there currently doesn't
  require softirq or irq safe context. If that needs to change,
  that's fine but that's a completely different reason than given
  above.

Thanks.

--
tejun

Subject: Re: [PATCH v3 12/13] execute the whole memcg freeing in rcu callback
Posted by Glauber Costa on Mon, 24 Sep 2012 08:48:01 GMT
View Forum Message <> Reply to Message

> And the above description too makes me scratch my head quite a bit. I
> can see what the patch is doing but can't understand the why.
>
> * Why was it punting the freeing to workqueue anyway? ISTR something
>   about static_keys but my memory fails. What changed? Why don't we
>   need it anymore?
>
> * As for locking context, the above description seems a bit misleading
>   to me. Synchronization constructs involved there currently doesn't
>   require softirq or irq safe context. If that needs to change,
>   that's fine but that's a completely different reason than given

> above.
>
> Thanks.
>

I just suck at changelogs =(

The problem here is very much like the one we had with static branches.
In that case, we had the problem with the cgroup_lock() being held, in
which case the jump label lock could not be called.

In here, after the kmem patches are in, the destruction function could
be called directly from memcg_kmem_uncharge_page() when the last put is
done. But this can actually be called from the page allocator, with an
incompatible softirq context. So it is not that it could be called, they
are actually called in that context at this point.

Subject: Re: [PATCH v3 06/13] memcg: kmem controller infrastructure
Posted by Michal Hocko on Wed, 26 Sep 2012 15:51:09 GMT
View Forum Message <> Reply to Message

On Tue 18-09-12 18:04:03, Glauber Costa wrote:
> This patch introduces infrastructure for tracking kernel memory pages to
> a given memcg. This will happen whenever the caller includes the flag
> __GFP_KMEMCG flag, and the task belong to a memcg other than the root.
>
> In memcontrol.h those functions are wrapped in inline acessors. The
> idea is to later on, patch those with static branches, so we don't incur
> any overhead when no mem cgroups with limited kmem are being used.

Could you describe the API a bit here, please? I guess the
kernel user is supposed to call memcg_kmem_newpage_charge and
memcg_kmem_commit_charge resp. memcg_kmem_uncharge_page.
All other kmem functions here are just helpers, right?
>
> [v2: improved comments and standardized function names]
> [v3: handle no longer opaque, functions not exported,
> even more comments]
> [v4: reworked Used bit handling and surroundings for more clarity]
>
> 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 | 97 ++++++++++++++++++++++++++
> mm/memcontrol.c | 177 ++
> 2 files changed, 274 insertions(+)
>
> diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
> index 8d9489f..82ede9a 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 +400,17 @@ struct sock;
> #ifdef CONFIG_MEMCG_KMEM
> void sock_update_memcg(struct sock *sk);
> void sock_release_memcg(struct sock *sk);
> +
> +static inline bool memcg_kmem_enabled(void)
> +{
> + return true;
> +}
> +
> +extern bool __memcg_kmem_newpage_charge(gfp_t gfp, struct mem_cgroup **memcg,
> +   int order);
> +extern void __memcg_kmem_commit_charge(struct page *page,
> +   struct mem_cgroup *memcg, int order);
> +extern void __memcg_kmem_uncharge_page(struct page *page, int order);
> #else
> static inline void sock_update_memcg(struct sock *sk)
> {
> @@ -406,6 +418,91 @@ static inline void sock_update_memcg(struct sock *sk)
> static inline void sock_release_memcg(struct sock *sk)
> {
> }
> +
> +static inline bool memcg_kmem_enabled(void)
> +{
> + return false;
> +}
> +
> +static inline bool
> +__memcg_kmem_newpage_charge(gfp_t gfp, struct mem_cgroup **memcg, int order)
> +{
> + return false;

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

I think we shouldn't care about these for !MEMCG_KMEM. It should be
sufficient to define the main three functions bellow as return true
resp. NOOP. This would reduce the code churn a bit and also make it
better maintainable.

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

> +}
> +
> +/**
> + * memcg_kmem_uncharge_page: uncharge pages from memcg
> + * @page: pointer to struct page being freed
> + * @order: allocation order.
> + *
> + * there is no need to specify memcg here, since it is embedded in page_cgroup
> + */
> +static __always_inline void
> +memcg_kmem_uncharge_page(struct page *page, int order)
> +{
> + if (memcg_kmem_enabled())
> + __memcg_kmem_uncharge_page(page, order);
> +}
> +
> +/**
> + * memcg_kmem_commit_charge: embeds correct memcg in a page
> + * @memcg: a pointer to the memcg this was charged against.
     ^^^^^^^
remove this one?

> + * @page: pointer to struct page recently allocated
> + * @memcg: the memcg structure we charged against
> + * @order: allocation order.
> + *
> + * Needs to be called after memcg_kmem_newpage_charge, regardless of success or
> + * failure of the allocation. if @page is NULL, this function will revert the
> + * charges. Otherwise, it will commit the memcg given by @memcg to the
> + * corresponding page_cgroup.
> + */
> +static __always_inline void
> +memcg_kmem_commit_charge(struct page *page, struct mem_cgroup *memcg, int order)
> +{
> + if (memcg_kmem_enabled() && memcg)
> + __memcg_kmem_commit_charge(page, memcg, order);
> +}
> #endif /* _LINUX_MEMCONTROL_H */
>
> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
> index f3fd354..0f36a01 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
> @@ -426,6 +430,9 @@ 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, u64 size);
> +static void memcg_uncharge_kmem(struct mem_cgroup *memcg, u64 size);
> +

Why the forward declarations here? We can simply move definitions up
before they are used for the first time, can't we? Besides that they are
never used/defined from outside of KMEM_MEMCG.

> void sock_update_memcg(struct sock *sk)
> {
>   if (mem_cgroup_sockets_enabled) {
> @@ -480,6 +487,110 @@ struct cg_proto *tcp_proto_cgroup(struct mem_cgroup *memcg)
> }
> EXPORT_SYMBOL(tcp_proto_cgroup);
> #endif /* CONFIG_INET */
> +
> +static inline bool memcg_can_account_kmem(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. We could also return the memcg directly and avoid the pointer
> + * passing, but a boolean return value gives better semantics considering
> + * the compiled-out case as well.
> + *
> + * Returning true means the allocation is possible.
> + */
> +bool

> +__memcg_kmem_newpage_charge(gfp_t gfp, struct mem_cgroup **_memcg, int order)
> +{
> + struct mem_cgroup *memcg;
> + bool ret;
> + struct task_struct *p;

Johannes likes christmas trees ;) and /me would like to remove `p' and
use mem_cgroup_from_task(rcu_dereference(current->mm->owner)) same as we
do at other places (I guess it will be checkpatch safe).

> +
> + *_memcg = NULL;
> + rcu_read_lock();
> + p = rcu_dereference(current->mm->owner);
> + memcg = mem_cgroup_from_task(p);

mem_cgroup_from_task says it can return NULL. Do we care here? If not
then please put VM_BUG_ON(!memcg) here.

> + rcu_read_unlock();
> +
> + if (!memcg_can_account_kmem(memcg))
> + return true;
> +
> + mem_cgroup_get(memcg);

I am confused. Why do we take a reference to memcg rather than css_get
here? Ahh it is because we keep the reference while the page is
allocated, right? Comment please.

I am still not sure whether we need css_get here as well. How do you
know that the current is not moved in parallel and it is a last task in
a group which then can go away?

> +
> + ret = memcg_charge_kmem(memcg, gfp, PAGE_SIZE << order) == 0;
> + if (ret)
> + *_memcg = memcg;
> + else
> + mem_cgroup_put(memcg);
> +
> + return ret;
> +}
> +
> +void __memcg_kmem_commit_charge(struct page *page, struct mem_cgroup *memcg,
> +   int order)
> +{
> + struct page_cgroup *pc;

> +
> + WARN_ON(mem_cgroup_is_root(memcg));

Why the warn? Nobody should use this directly and
memcg_kmem_commit_charge takes care of the root same as
__memcg_kmem_newpage_charge does. If it is for correctness then it
should be VM_BUG_ON.

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

disabling cannot happen, right?

> + *
> + * 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));

Same as above I do not see a reason for warn here. It just adds a code
and if you want it for debugging then VM_BUG_ON sounds more appropriate.
/me thinks

> + memcg_uncharge_kmem(memcg, PAGE_SIZE << order);
> + mem_cgroup_put(memcg);
> +}
> #endif /* CONFIG_MEMCG_KMEM */
>
> #if defined(CONFIG_INET) && defined(CONFIG_MEMCG_KMEM)
> @@ -5700,3 +5811,69 @@ 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, u64 size)
> +{
> + 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);

A comment please? Why __GFP_IO is not considered for example?

> +
> + ret = 0;
> +
> + if (!memcg)
> + return ret;

How can we get a NULL memcg here without blowing in
__memcg_kmem_newpage_charge?

```
> +
> + _memcg = memcg;
> + ret = __mem_cgroup_try_charge(NULL, gfp, size / PAGE_SIZE,

me likes >> PAGE_SHIFT more.

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

You could save few lines and get rid of the strange nofail by:
[...]
+  res_counter_charge_nofail(&memcg->res, size, &fail_res);
+  if (do_swap_account)
+   res_counter_charge_nofail(&memcg->memsw, size,
+         &fail_res);
+  res_counter_charge_nofail(&memcg->kmem, size, &fail_res);
+  return 0;
+ } else if (ret == -ENOMEM)
```

```
+  return ret;
+ else
+  ret = res_counter_charge(&memcg->kmem, size, &fail_res);
+
+ if (ret) {
+  res_counter_uncharge(&memcg->res, size);
+  if (do_swap_account)
+   res_counter_uncharge(&memcg->memsw, size);
+ }

> +
> + return ret;
> +}
> +
> +void memcg_uncharge_kmem(struct mem_cgroup *memcg, u64 size)
> +{
> + if (!memcg)
> +  return;
> +
> + res_counter_uncharge(&memcg->kmem, size);
> + res_counter_uncharge(&memcg->res, size);
> + if (do_swap_account)
> +  res_counter_uncharge(&memcg->memsw, size);
> +}
> +#endif /* CONFIG_MEMCG_KMEM */
> --
> 1.7.11.4
>
> --
> 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 v3 06/13] memcg: kmem controller infrastructure
Posted by Glauber Costa on Thu, 27 Sep 2012 11:31:57 GMT
View Forum Message <> Reply to Message

On 09/26/2012 07:51 PM, Michal Hocko wrote:
> On Tue 18-09-12 18:04:03, Glauber Costa wrote:
>> This patch introduces infrastructure for tracking kernel memory pages to
>> a given memcg. This will happen whenever the caller includes the flag
>> __GFP_KMEMCG flag, and the task belong to a memcg other than the root.
>>

>> In memcontrol.h those functions are wrapped in inline acessors. The
>> idea is to later on, patch those with static branches, so we don't incur
>> any overhead when no mem cgroups with limited kmem are being used.
>
> Could you describe the API a bit here, please? I guess the
> kernel user is supposed to call memcg_kmem_newpage_charge and
> memcg_kmem_commit_charge resp. memcg_kmem_uncharge_page.
> All other kmem functions here are just helpers, right?
Yes, sir.

>>
>> [v2: improved comments and standardized function names]
>> [v3: handle no longer opaque, functions not exported,
>>   even more comments]
>> [v4: reworked Used bit handling and surroundings for more clarity]
>>
>> 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 | 97 +++++++++++++++++++++++++
>> mm/memcontrol.c | 177 +++
>> 2 files changed, 274 insertions(+)
>>
>> diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
>> index 8d9489f..82ede9a 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 +400,17 @@ struct sock;
>> #ifdef CONFIG_MEMCG_KMEM
>> void sock_update_memcg(struct sock *sk);
>> void sock_release_memcg(struct sock *sk);
>> +
>> +static inline bool memcg_kmem_enabled(void)
>> +{
>> + return true;
>> +}

>> +
>> +extern bool __memcg_kmem_newpage_charge(gfp_t gfp, struct mem_cgroup **memcg,
>> +   int order);
>> +extern void __memcg_kmem_commit_charge(struct page *page,
>> +   struct mem_cgroup *memcg, int order);
>> +extern void __memcg_kmem_uncharge_page(struct page *page, int order);
>> #else
>> static inline void sock_update_memcg(struct sock *sk)
>> {
>> @@ -406,6 +418,91 @@ static inline void sock_update_memcg(struct sock *sk)
>> static inline void sock_release_memcg(struct sock *sk)
>> {
>> }
>> +
>> +static inline bool memcg_kmem_enabled(void)
>> +{
>> + return false;
>> +}
>> +
>> +static inline bool
>> +__memcg_kmem_newpage_charge(gfp_t gfp, struct mem_cgroup **memcg, int order)
>> +{
>> + return false;
>> +}
>> +
>> +static inline void __memcg_kmem_uncharge_page(struct page *page, int order)
>> +{
>> +}
>> +
>> +static inline void
>> +__memcg_kmem_commit_charge(struct page *page, struct mem_cgroup *memcg, int order)
>> +{
>> +}
>
> I think we shouldn't care about these for !MEMCG_KMEM. It should be
> sufficient to define the main three functions bellow as return true
> resp. NOOP. This would reduce the code churn a bit and also make it
> better maintainable.
>

Ok.

>> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
>> index f3fd354..0f36a01 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
>> @@ -426,6 +430,9 @@ 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, u64 size);
>> +static void memcg_uncharge_kmem(struct mem_cgroup *memcg, u64 size);
>> +
>
> Why the forward declarations here? We can simply move definitions up
> before they are used for the first time, can't we? Besides that they are
> never used/defined from outside of KMEM_MEMCG.
>
I see your point, given the recent patch about gcc complaining about
those things. Will change.

>> +
>> + *_memcg = NULL;
>> + rcu_read_lock();
>> + p = rcu_dereference(current->mm->owner);
>> + memcg = mem_cgroup_from_task(p);
>
> mem_cgroup_from_task says it can return NULL. Do we care here? If not
> then please put VM_BUG_ON(!memcg) here.
>
>> + rcu_read_unlock();
>> +
>> + if (!memcg_can_account_kmem(memcg))
>> + return true;
>> +
>> + mem_cgroup_get(memcg);
>
> I am confused. Why do we take a reference to memcg rather than css_get
> here? Ahh it is because we keep the reference while the page is
> allocated, right? Comment please.
ok.

>
> I am still not sure whether we need css_get here as well. How do you

> know that the current is not moved in parallel and it is a last task in
> a group which then can go away?

the reference count aquired by mem_cgroup_get will still prevent the
memcg from going away, no?

```
>> +
>> + /* The page allocation failed. Revert */
>> + if (!page) {
>> +  memcg_uncharge_kmem(memcg, PAGE_SIZE << order);
>> +  return;
>> + }
>> +
>> + pc = lookup_page_cgroup(page);
>> + lock_page_cgroup(pc);
>> + pc->mem_cgroup = memcg;
>> + SetPageCgroupUsed(pc);
>> + unlock_page_cgroup(pc);
>> +}
>> +
>> +void __memcg_kmem_uncharge_page(struct page *page, int order)
>> +{
>> + struct mem_cgroup *memcg = NULL;
>> + struct page_cgroup *pc;
>> +
>> +
>> + pc = lookup_page_cgroup(page);
>> + /*
>> +  * Fast unlocked return. Theoretically might have changed, have to
>> +  * check again after locking.
>> +  */
>> + if (!PageCgroupUsed(pc))
>> +  return;
>> +
>> + lock_page_cgroup(pc);
>> + if (PageCgroupUsed(pc)) {
>> +  memcg = pc->mem_cgroup;
>> +  ClearPageCgroupUsed(pc);
>> + }
>> + unlock_page_cgroup(pc);
>> +
>> + /*
>> +  * 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.
>
> disabling cannot happen, right?
>
```

not anymore, right. I can update the comment, but I still believe it is
a lot saner to trust information in page_cgroup.

>> +#ifdef CONFIG_MEMCG_KMEM
>> +int memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t gfp, u64 size)
>> +{
>> + 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);
>
> A comment please? Why __GFP_IO is not considered for example?
>
>

Actually, I believe testing for GFP_WAIT and !GFP_NORETRY would be enough.

The rationale here is, of course, under which circumstance would it be
valid to call the oom killer? Which is, if the allocation can wait, and
can retry.

>
> You could save few lines and get rid of the strange nofail by:
> [...]
> + res_counter_charge_nofail(&memcg->res, size, &fail_res);
> + if (do_swap_account)
> + res_counter_charge_nofail(&memcg->memsw, size,
> + &fail_res);
> + res_counter_charge_nofail(&memcg->kmem, size, &fail_res);
> + return 0;
> + } else if (ret == -ENOMEM)
> + return ret;
> + else
> + ret = res_counter_charge(&memcg->kmem, size, &fail_res);
> +
> + if (ret) {
> + res_counter_uncharge(&memcg->res, size);
> + if (do_swap_account)
> + res_counter_uncharge(&memcg->memsw, size);
> + }
>
indeed.

Subject: Re: [PATCH v3 05/13] Add a __GFP_KMEMCG flag
Posted by Mel Gorman on Thu, 27 Sep 2012 13:34:35 GMT
View Forum Message <> Reply to Message

On Tue, Sep 18, 2012 at 06:04:02PM +0400, 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: 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>
> Acked-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>

I agree with Christophs recommendation that this flag always exist instead
of being 0 in the !MEMCG_KMEM case. If __GFP_KMEMCG ever is used in another
part of the VM (which would be unexpected but still) then the behaviour
might differ too much between MEMCG_KMEM and !MEMCG_KMEM cases. As unlikely
as the case is, it's not impossible.

For tracing __GFP_KMEMCG should have an entry in
include/trace/events/gfpflags.h

Get rid of the CONFIG_MEMCG_KMEM check and update
include/trace/events/gfpflags.h and then feel free to stick my Acked-by
on it.

--
Mel Gorman
SUSE Labs

Subject: Re: [PATCH v3 05/13] Add a __GFP_KMEMCG flag
Posted by Glauber Costa on Thu, 27 Sep 2012 13:41:05 GMT
View Forum Message <> Reply to Message

On 09/27/2012 05:34 PM, Mel Gorman wrote:
> On Tue, Sep 18, 2012 at 06:04:02PM +0400, 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: 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>
>> Acked-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
>
> I agree with Christophs recommendation that this flag always exist instead
> of being 0 in the !MEMCG_KMEM case. If __GFP_KMEMCG ever is used in another
> part of the VM (which would be unexpected but still) then the behaviour
> might differ too much between MEMCG_KMEM and !MEMCG_KMEM cases. As unlikely
> as the case is, it's not impossible.
>
> For tracing __GFP_KMEMCG should have an entry in
> include/trace/events/gfpflags.h
>
> Get rid of the CONFIG_MEMCG_KMEM check and update
> include/trace/events/gfpflags.h and then feel free to stick my Acked-by
> on it.
>

Thanks, that is certainly doable.

Subject: Re: [PATCH v3 06/13] memcg: kmem controller infrastructure
Posted by Michal Hocko on Thu, 27 Sep 2012 13:44:32 GMT
View Forum Message <> Reply to Message

On Thu 27-09-12 15:31:57, Glauber Costa wrote:
> On 09/26/2012 07:51 PM, Michal Hocko wrote:
> > On Tue 18-09-12 18:04:03, Glauber Costa wrote:
[...]
> >> + *_memcg = NULL;
> >> + rcu_read_lock();
> >> + p = rcu_dereference(current->mm->owner);
> >> + memcg = mem_cgroup_from_task(p);
> >
> > mem_cgroup_from_task says it can return NULL. Do we care here? If not
> > then please put VM_BUG_ON(!memcg) here.
> >
> >> + rcu_read_unlock();
> >> +
> >> + if (!memcg_can_account_kmem(memcg))
> >> + return true;

> >> +
> >> + mem_cgroup_get(memcg);
> >
> > I am confused. Why do we take a reference to memcg rather than css_get
> > here? Ahh it is because we keep the reference while the page is
> > allocated, right? Comment please.
> ok.
>
> >
> > I am still not sure whether we need css_get here as well. How do you
> > know that the current is not moved in parallel and it is a last task in
> > a group which then can go away?
>
> the reference count aquired by mem_cgroup_get will still prevent the
> memcg from going away, no?

Yes but you are outside of the rcu now and we usually do css_get before
we rcu_unlock. mem_cgroup_get just makes sure the group doesn't get
deallocated but it could be gone before you call it. Or I am just
confused - these 2 levels of ref counting is really not nice.

Anyway, I have just noticed that __mem_cgroup_try_charge does
VM_BUG_ON(css_is_removed(&memcg->css)) on a given memcg so you should
keep css ref count up as well.

> >> + /* The page allocation failed. Revert */
> >> + if (!page) {
> >> + memcg_uncharge_kmem(memcg, PAGE_SIZE << order);
> >> + return;
> >> + }
> >> +
> >> + pc = lookup_page_cgroup(page);
> >> + lock_page_cgroup(pc);
> >> + pc->mem_cgroup = memcg;
> >> + SetPageCgroupUsed(pc);
> >> + unlock_page_cgroup(pc);
> >> +}
> >> +
> >> +void __memcg_kmem_uncharge_page(struct page *page, int order)
> >> +{
> >> + struct mem_cgroup *memcg = NULL;
> >> + struct page_cgroup *pc;
> >> +
> >> +
> >> + pc = lookup_page_cgroup(page);
> >> + /*
> >> + * Fast unlocked return. Theoretically might have changed, have to
> >> + * check again after locking.

> >> + */
> >> + if (!PageCgroupUsed(pc))
> >> + return;
> >> +
> >> + lock_page_cgroup(pc);
> >> + if (PageCgroupUsed(pc)) {
> >> + memcg = pc->mem_cgroup;
> >> + ClearPageCgroupUsed(pc);
> >> + }
> >> + unlock_page_cgroup(pc);
> >> +
> >> + /*
> >> + * 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.
> >
> > disabling cannot happen, right?
> >
> not anymore, right. I can update the comment,

yes, it is confusing

> but I still believe it is a lot saner to trust information in
> page_cgroup.

I have no objections against that. PageCgroupUsed test and using
pc->mem_cgroup is fine.

> >> +#ifdef CONFIG_MEMCG_KMEM
> >> +int memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t gfp, u64 size)
> >> +{
> >> + 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);
> >
> > A comment please? Why __GFP_IO is not considered for example?
> >
> >
>
> Actually, I believe testing for GFP_WAIT and !GFP_NORETRY would be enough.
>
> The rationale here is, of course, under which circumstance would it be
> valid to call the oom killer? Which is, if the allocation can wait, and

> can retry.

Yes __GFP_WAIT is clear because memcg OOM can wait for arbitrary amount
of time (wait for userspace action on oom_control). __GFP_NORETRY
couldn't get to oom before because oom was excluded explicitely for THP
and migration didn't go through the charging path to reach the oom.
But I do agree that __GFP_NORETRY allocations shouldn't cause the OOM
because we should rather fail the allocation from kernel rather than
shoot something.

--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v3 07/13] mm: Allocate kernel pages to the right memcg
Posted by Mel Gorman on Thu, 27 Sep 2012 13:50:53 GMT
View Forum Message <> Reply to Message

On Tue, Sep 18, 2012 at 06:04:04PM +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,
> whoever is marking its allocations with __GFP_KMEMCG flag is responsible
> for telling the page allocator that this is such an allocation at
> free_pages() time. This is done by the invocation of
> __free_accounted_pages() and free_accounted_pages().
>
> [v2: inverted test order to avoid a memcg_get leak,
>   free_accounted_pages simplification]
>
> 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: Johannes Weiner <hannes@cmpxchg.org>
> CC: Suleiman Souhlal <suleiman@google.com>
> CC: Mel Gorman <mgorman@suse.de>
> Acked-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
> ---
> include/linux/gfp.h | 3 +++
> mm/page_alloc.c | 35 +++++++++++++++++++++++++++++++++++
> 2 files changed, 38 insertions(+)
>
> diff --git a/include/linux/gfp.h b/include/linux/gfp.h

> index 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 b0c5a52..897d8e2 100644
> --- a/mm/page_alloc.c
> +++ b/mm/page_alloc.c
> @@ -2573,6 +2573,7 @@ __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;
> + struct mem_cgroup *memcg = NULL;
>
>   gfp_mask &= gfp_allowed_mask;
>
> @@ -2591,6 +2592,13 @@ __alloc_pages_nodemask(gfp_t gfp_mask, unsigned int order,
>   if (unlikely(!zonelist->_zonerefs->zone))
>   return NULL;
>
> + /*
> + * Will only have any effect when __GFP_KMEMCG is set. This is
> + * verified in the (always inline) callee
> + */
> + if (!memcg_kmem_newpage_charge(gfp_mask, &memcg, order))
> + return NULL;
> +

1. returns quickly if memcg disabled
2. returns quickly if !__GFP_KMEMCG
3. returns quickly for kernel threads and interrupts

I'm expecting that these paths are completely dead when kmem accounting
is off so

Acked-by: Mel Gorman <mgorman@suse.de>

That said, it's not directly related to this patch but I would suggest that

you also check for TIF_MEMDIE in memcg_kmem_newpage_charge. It would be very
silly if a process failed to exit because it couldn't allocate a page it
needed. I expect that such a case is impossible today but it might change in
the future. If you're doing another revision, an extra check would not hurt.

It's difficult to predict if it should be making all the checks that
gfp_to_alloc_flags() does but you might need to in the future so keep it
in mind.

> retry_cpuset:
>   cpuset_mems_cookie = get_mems_allowed();
>
> @@ -2624,6 +2632,8 @@ out:
>   if (unlikely(!put_mems_allowed(cpuset_mems_cookie) && !page))
>   goto retry_cpuset;
>
> + memcg_kmem_commit_charge(page, memcg, order);
> +
>   return page;

Ok.

> }
> EXPORT_SYMBOL(__alloc_pages_nodemask);
> @@ -2676,6 +2686,31 @@ void free_pages(unsigned long addr, unsigned int order)
>
> EXPORT_SYMBOL(free_pages);
>
> +/*
> + * __free_accounted_pages and free_accounted_pages will free pages allocated
> + * with __GFP_KMEMCG.
> + *
> + * Those pages are accounted to a particular memcg, embedded in the
> + * corresponding page_cgroup. To avoid adding a hit in the allocator to search
> + * for that information only to find out that it is NULL for users who have no
> + * interest in that whatsoever, we provide these functions.
> + *
> + * The caller knows better which flags it relies on.
> + */
> +void __free_accounted_pages(struct page *page, unsigned int order)
> +{
> + memcg_kmem_uncharge_page(page, order);
> + __free_pages(page, order);
> +}
> +

> +void free_accounted_pages(unsigned long addr, unsigned int order)
> +{
> + if (addr != 0) {
> + VM_BUG_ON(!virt_addr_valid((void *)addr));

This is probably overkill. If it's invalid, the next line is likely to
blow up anyway. It's no biggie.

> + __free_accounted_pages(virt_to_page((void *)addr), order);
> + }
> +}
> +
> static void *make_alloc_exact(unsigned long addr, unsigned order, size_t size)
> {
>   if (addr) {
> --
> 1.7.11.4
>

--
Mel Gorman
SUSE Labs

Subject: Re: [PATCH v3 07/13] mm: Allocate kernel pages to the right memcg
Posted by Michal Hocko on Thu, 27 Sep 2012 13:52:39 GMT
View Forum Message <> Reply to Message

On Tue 18-09-12 18:04:04, Glauber Costa wrote:
> When a process tries to allocate a page with the __GFP_KMEMCG flag, the
> page allocator will call the corresponding memcg functions to validate
> the allocation. Tasks in the root memcg can always proceed.
>
> To avoid adding markers to the page - and a kmem flag that would
> necessarily follow, as much as doing page_cgroup lookups for no reason,
> whoever is marking its allocations with __GFP_KMEMCG flag is responsible
> for telling the page allocator that this is such an allocation at
> free_pages() time. This is done by the invocation of
> __free_accounted_pages() and free_accounted_pages().
>
> [v2: inverted test order to avoid a memcg_get leak,
>   free_accounted_pages simplification]
>
> 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: Johannes Weiner <hannes@cmpxchg.org>

> CC: Suleiman Souhlal <suleiman@google.com>
> CC: Mel Gorman <mgorman@suse.de>
> Acked-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>

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

Thanks!

> ---
> include/linux/gfp.h | 3 +++
> mm/page_alloc.c | 35 +++++++++++++++++++++++++++++++++++
> 2 files changed, 38 insertions(+)
>
> diff --git a/include/linux/gfp.h b/include/linux/gfp.h
> index 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 b0c5a52..897d8e2 100644
> --- a/mm/page_alloc.c
> +++ b/mm/page_alloc.c
> @@ -2573,6 +2573,7 @@ __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;
> + struct mem_cgroup *memcg = NULL;
>
>   gfp_mask &= gfp_allowed_mask;
>
> @@ -2591,6 +2592,13 @@ __alloc_pages_nodemask(gfp_t gfp_mask, unsigned int order,
>   if (unlikely(!zonelist->_zonerefs->zone))
>   return NULL;
>
> + /*
> + * Will only have any effect when __GFP_KMEMCG is set. This is
> + * verified in the (always inline) callee
> + */
> + if (!memcg_kmem_newpage_charge(gfp_mask, &memcg, order))

```
> +  return NULL;
> +
>  retry_cpuset:
>   cpuset_mems_cookie = get_mems_allowed();
>
> @@ -2624,6 +2632,8 @@ out:
>   if (unlikely(!put_mems_allowed(cpuset_mems_cookie) && !page))
>    goto retry_cpuset;
>
> + memcg_kmem_commit_charge(page, memcg, order);
> +
>   return page;
>  }
>  EXPORT_SYMBOL(__alloc_pages_nodemask);
> @@ -2676,6 +2686,31 @@ void free_pages(unsigned long addr, unsigned int order)
>
>  EXPORT_SYMBOL(free_pages);
>
> +/*
> + * __free_accounted_pages and free_accounted_pages will free pages allocated
> + * with __GFP_KMEMCG.
> + *
> + * Those pages are accounted to a particular memcg, embedded in the
> + * corresponding page_cgroup. To avoid adding a hit in the allocator to search
> + * for that information only to find out that it is NULL for users who have no
> + * interest in that whatsoever, we provide these functions.
> + *
> + * The caller knows better which flags it relies on.
> + */
> +void __free_accounted_pages(struct page *page, unsigned int order)
> +{
> + memcg_kmem_uncharge_page(page, order);
> + __free_pages(page, order);
> +}
> +
> +void free_accounted_pages(unsigned long addr, unsigned int order)
> +{
> + if (addr != 0) {
> +  VM_BUG_ON(!virt_addr_valid((void *)addr));
> +  __free_accounted_pages(virt_to_page((void *)addr), order);
> + }
> +}
> +
>  static void *make_alloc_exact(unsigned long addr, unsigned order, size_t size)
>  {
>   if (addr) {
> --
> 1.7.11.4
```

>
> --
> 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 v3 07/13] mm: Allocate kernel pages to the right memcg
Posted by Glauber Costa on Fri, 28 Sep 2012 09:43:47 GMT
View Forum Message <> Reply to Message

On 09/27/2012 05:50 PM, Mel Gorman wrote:
>> +void __free_accounted_pages(struct page *page, unsigned int order)
>> > +{
>> > + memcg_kmem_uncharge_page(page, order);
>> > + __free_pages(page, order);
>> > +}
>> > +
>> > +void free_accounted_pages(unsigned long addr, unsigned int order)
>> > +{
>> > + if (addr != 0) {
>> > + VM_BUG_ON(!virt_addr_valid((void *)addr));
> This is probably overkill. If it's invalid, the next line is likely to
> blow up anyway. It's no biggie.
>

So this is here because it is in free_pages() as well. If it blows, at
least we know precisely why (if debugging), and VM_BUG_ON() is only
compiled in when CONFIG_DEBUG_VM.

But I'm fine with either.
Should it stay or should it go ?

Subject: Re: [PATCH v3 06/13] memcg: kmem controller infrastructure
Posted by Glauber Costa on Fri, 28 Sep 2012 11:34:19 GMT
View Forum Message <> Reply to Message

On 09/27/2012 05:44 PM, Michal Hocko wrote:
>> > the reference count aquired by mem_cgroup_get will still prevent the
>> > memcg from going away, no?
> Yes but you are outside of the rcu now and we usually do css_get before
> we rcu_unlock. mem_cgroup_get just makes sure the group doesn't get

> deallocated but it could be gone before you call it. Or I am just
> confused - these 2 levels of ref counting is really not nice.
>
> Anyway, I have just noticed that __mem_cgroup_try_charge does
> VM_BUG_ON(css_is_removed(&memcg->css)) on a given memcg so you should
> keep css ref count up as well.
>

IIRC, css_get will prevent the cgroup directory from being removed.
Because some allocations are expected to outlive the cgroup, we
specifically don't want that.

Subject: Re: [PATCH v3 07/13] mm: Allocate kernel pages to the right memcg
Posted by Mel Gorman on Fri, 28 Sep 2012 13:28:39 GMT
View Forum Message <> Reply to Message

On Fri, Sep 28, 2012 at 01:43:47PM +0400, Glauber Costa wrote:
> On 09/27/2012 05:50 PM, Mel Gorman wrote:
> >> +void __free_accounted_pages(struct page *page, unsigned int order)
> >> > +{
> >> > + memcg_kmem_uncharge_page(page, order);
> >> > + __free_pages(page, order);
> >> > +}
> >> > +
> >> > +void free_accounted_pages(unsigned long addr, unsigned int order)
> >> > +{
> >> > + if (addr != 0) {
> >> > + VM_BUG_ON(!virt_addr_valid((void *)addr));
> > This is probably overkill. If it's invalid, the next line is likely to
> > blow up anyway. It's no biggie.
> >
>
> So this is here because it is in free_pages() as well. If it blows, at
> least we know precisely why (if debugging), and VM_BUG_ON() is only
> compiled in when CONFIG_DEBUG_VM.
>

Ah, I see.

> But I'm fine with either.
> Should it stay or should it go ?
>

It can stay. It makes sense that it look similar to free_pages() and as
you say, it makes debugging marginally easier.

--
Mel Gorman
SUSE Labs

Subject: Re: [PATCH v3 06/13] memcg: kmem controller infrastructure
Posted by Tejun Heo on Sun, 30 Sep 2012 08:25:42 GMT
View Forum Message <> Reply to Message

On Fri, Sep 28, 2012 at 03:34:19PM +0400, Glauber Costa wrote:
> On 09/27/2012 05:44 PM, Michal Hocko wrote:
> > Anyway, I have just noticed that __mem_cgroup_try_charge does
> > VM_BUG_ON(css_is_removed(&memcg->css)) on a given memcg so you should
> > keep css ref count up as well.
>
> IIRC, css_get will prevent the cgroup directory from being removed.
> Because some allocations are expected to outlive the cgroup, we
> specifically don't want that.

That synchronous ref draining is going away. Maybe we can do that
before kmemcg? Michal, do you have some timeframe on mind?

Thanks.

--
tejun

Subject: Re: [PATCH v3 06/13] memcg: kmem controller infrastructure
Posted by Glauber Costa on Mon, 01 Oct 2012 08:28:28 GMT
View Forum Message <> Reply to Message

On 09/30/2012 12:25 PM, Tejun Heo wrote:
> On Fri, Sep 28, 2012 at 03:34:19PM +0400, Glauber Costa wrote:
>> On 09/27/2012 05:44 PM, Michal Hocko wrote:
>>> Anyway, I have just noticed that __mem_cgroup_try_charge does
>>> VM_BUG_ON(css_is_removed(&memcg->css)) on a given memcg so you should
>>> keep css ref count up as well.
>>
>> IIRC, css_get will prevent the cgroup directory from being removed.
>> Because some allocations are expected to outlive the cgroup, we
>> specifically don't want that.
>
> That synchronous ref draining is going away. Maybe we can do that
> before kmemcg? Michal, do you have some timeframe on mind?
>

Since you said yourself in other points in this thread that you are fine
with some page references outliving the cgroup in the case of slab, this
is a situation that comes with the code, not a situation that was
incidentally there, and we're making use of.

Subject: Re: [PATCH v3 06/13] memcg: kmem controller infrastructure
Posted by Michal Hocko on Mon, 01 Oct 2012 09:44:47 GMT
View Forum Message <> Reply to Message

On Sun 30-09-12 17:25:42, Tejun Heo wrote:
> On Fri, Sep 28, 2012 at 03:34:19PM +0400, Glauber Costa wrote:
> > On 09/27/2012 05:44 PM, Michal Hocko wrote:
> > > Anyway, I have just noticed that __mem_cgroup_try_charge does
> > > VM_BUG_ON(css_is_removed(&memcg->css)) on a given memcg so you should
> > > keep css ref count up as well.
> >
> > IIRC, css_get will prevent the cgroup directory from being removed.
> > Because some allocations are expected to outlive the cgroup, we
> > specifically don't want that.
>
> That synchronous ref draining is going away. Maybe we can do that
> before kmemcg? Michal, do you have some timeframe on mind?

It is on my todo list but I didn't get to it yet. I am not sure we can
get rid of css_get though - will have to think about that.

>
> Thanks.
>
> --
> tejun

--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v3 06/13] memcg: kmem controller infrastructure
Posted by Michal Hocko on Mon, 01 Oct 2012 09:48:46 GMT
View Forum Message <> Reply to Message

On Fri 28-09-12 15:34:19, Glauber Costa wrote:
> On 09/27/2012 05:44 PM, Michal Hocko wrote:
> >> > the reference count aquired by mem_cgroup_get will still prevent the
> >> > memcg from going away, no?
> > Yes but you are outside of the rcu now and we usually do css_get before

> > we rcu_unlock. mem_cgroup_get just makes sure the group doesn't get
> > deallocated but it could be gone before you call it. Or I am just
> > confused - these 2 levels of ref counting is really not nice.
> >
> > Anyway, I have just noticed that __mem_cgroup_try_charge does
> > VM_BUG_ON(css_is_removed(&memcg->css)) on a given memcg so you should
> > keep css ref count up as well.
> >
>
> IIRC, css_get will prevent the cgroup directory from being removed.
> Because some allocations are expected to outlive the cgroup, we
> specifically don't want that.

Yes, but how do you guarantee that the above VM_BUG_ON doesn't trigger?
Task could have been moved to another group between mem_cgroup_from_task
and mem_cgroup_get, no?
--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v3 06/13] memcg: kmem controller infrastructure
Posted by Glauber Costa on Mon, 01 Oct 2012 10:09:09 GMT
View Forum Message <> Reply to Message

On 10/01/2012 01:48 PM, Michal Hocko wrote:
> On Fri 28-09-12 15:34:19, Glauber Costa wrote:
>> On 09/27/2012 05:44 PM, Michal Hocko wrote:
>>>>> the reference count aquired by mem_cgroup_get will still prevent the
>>>>> memcg from going away, no?
>>> Yes but you are outside of the rcu now and we usually do css_get before
>>> we rcu_unlock. mem_cgroup_get just makes sure the group doesn't get
>>> deallocated but it could be gone before you call it. Or I am just
>>> confused - these 2 levels of ref counting is really not nice.
>>>
>>> Anyway, I have just noticed that __mem_cgroup_try_charge does
>>> VM_BUG_ON(css_is_removed(&memcg->css)) on a given memcg so you should
>>> keep css ref count up as well.
>>>
>>
>> IIRC, css_get will prevent the cgroup directory from being removed.
>> Because some allocations are expected to outlive the cgroup, we
>> specifically don't want that.
>
> Yes, but how do you guarantee that the above VM_BUG_ON doesn't trigger?
> Task could have been moved to another group between mem_cgroup_from_task
> and mem_cgroup_get, no?
>

Ok, after reading this again (and again), you seem to be right. It
concerns me, however, that simply getting the css would lead us to a
double get/put pair, since try_charge will have to do it anyway.

I considered just letting try_charge selecting the memcg, but that is
not really what we want, since if that memcg will fail kmem allocations,
we simply won't issue try charge, but return early.

Any immediate suggestions on how to handle this ?

Subject: Re: [PATCH v3 06/13] memcg: kmem controller infrastructure
Posted by Glauber Costa on Mon, 01 Oct 2012 11:51:20 GMT
View Forum Message <> Reply to Message

On 10/01/2012 03:51 PM, Michal Hocko wrote:
> On Mon 01-10-12 14:09:09, Glauber Costa wrote:
>> On 10/01/2012 01:48 PM, Michal Hocko wrote:
>>> On Fri 28-09-12 15:34:19, Glauber Costa wrote:
>>>> On 09/27/2012 05:44 PM, Michal Hocko wrote:
>>>>>> the reference count aquired by mem_cgroup_get will still prevent the
>>>>>> memcg from going away, no?
>>>>> Yes but you are outside of the rcu now and we usually do css_get before
>>>>> we rcu_unlock. mem_cgroup_get just makes sure the group doesn't get
>>>>> deallocated but it could be gone before you call it. Or I am just
>>>>> confused - these 2 levels of ref counting is really not nice.
>>>>>
>>>>> Anyway, I have just noticed that __mem_cgroup_try_charge does
>>>>> VM_BUG_ON(css_is_removed(&memcg->css)) on a given memcg so you should
>>>>> keep css ref count up as well.
>>>>>
>>>>
>>>> IIRC, css_get will prevent the cgroup directory from being removed.
>>>> Because some allocations are expected to outlive the cgroup, we
>>>> specifically don't want that.
>>>
>>> Yes, but how do you guarantee that the above VM_BUG_ON doesn't trigger?
>>> Task could have been moved to another group between mem_cgroup_from_task
>>> and mem_cgroup_get, no?
>>>
>>
>> Ok, after reading this again (and again), you seem to be right. It
>> concerns me, however, that simply getting the css would lead us to a
>> double get/put pair, since try_charge will have to do it anyway.
>
> That happens only for !*ptr case and you provide a memcg here, don't
> you.

>

```
if (*ptr) { /* css should be a valid one */
        memcg = *ptr;
        VM_BUG_ON(css_is_removed(&memcg->css));
        if (mem_cgroup_is_root(memcg))
                goto done;
        if (consume_stock(memcg, nr_pages))
                goto done;
        css_get(&memcg->css);
```

The way I read this, this will still issue a css_get here, unless
consume_stock suceeds (assuming non-root)

So we'd still have to have a wrapping get/put pair outside the charge.

Subject: Re: [PATCH v3 06/13] memcg: kmem controller infrastructure
Posted by Michal Hocko on Mon, 01 Oct 2012 11:51:57 GMT
View Forum Message <> Reply to Message

On Mon 01-10-12 14:09:09, Glauber Costa wrote:
> On 10/01/2012 01:48 PM, Michal Hocko wrote:
> > On Fri 28-09-12 15:34:19, Glauber Costa wrote:
> >> On 09/27/2012 05:44 PM, Michal Hocko wrote:
> >>>> the reference count aquired by mem_cgroup_get will still prevent the
> >>>> memcg from going away, no?
> >>> Yes but you are outside of the rcu now and we usually do css_get before
> >>> we rcu_unlock. mem_cgroup_get just makes sure the group doesn't get
> >>> deallocated but it could be gone before you call it. Or I am just
> >>> confused - these 2 levels of ref counting is really not nice.
> >>>
> >>> Anyway, I have just noticed that __mem_cgroup_try_charge does
> >>> VM_BUG_ON(css_is_removed(&memcg->css)) on a given memcg so you should
> >>> keep css ref count up as well.
> >>>
> >>
> >> IIRC, css_get will prevent the cgroup directory from being removed.
> >> Because some allocations are expected to outlive the cgroup, we
> >> specifically don't want that.
> >
> > Yes, but how do you guarantee that the above VM_BUG_ON doesn't trigger?
> > Task could have been moved to another group between mem_cgroup_from_task
> > and mem_cgroup_get, no?
> >
>
> Ok, after reading this again (and again), you seem to be right. It

> concerns me, however, that simply getting the css would lead us to a
> double get/put pair, since try_charge will have to do it anyway.

That happens only for !*ptr case and you provide a memcg here, don't
you.

> I considered just letting try_charge selecting the memcg, but that is
> not really what we want, since if that memcg will fail kmem allocations,
> we simply won't issue try charge, but return early.
>
> Any immediate suggestions on how to handle this ?

I would do the same thing __mem_cgroup_try_charge does.
retry:
```
 rcu_read_lock();
 p = rcu_dereference(mm->owner);
 if (!css_tryget(&memcg->css)) {
  rcu_read_unlock();
  goto retry;
 }
 rcu_read_unlock();
```

--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v3 06/13] memcg: kmem controller infrastructure
Posted by Michal Hocko on Mon, 01 Oct 2012 11:58:47 GMT
View Forum Message <> Reply to Message

On Mon 01-10-12 15:51:20, Glauber Costa wrote:
> On 10/01/2012 03:51 PM, Michal Hocko wrote:
> > On Mon 01-10-12 14:09:09, Glauber Costa wrote:
> >> On 10/01/2012 01:48 PM, Michal Hocko wrote:
> >>> On Fri 28-09-12 15:34:19, Glauber Costa wrote:
> >>>> On 09/27/2012 05:44 PM, Michal Hocko wrote:
> >>>>> the reference count aquired by mem_cgroup_get will still prevent the
> >>>>> memcg from going away, no?
> >>>> Yes but you are outside of the rcu now and we usually do css_get before
> >>>> we rcu_unlock. mem_cgroup_get just makes sure the group doesn't get
> >>>> deallocated but it could be gone before you call it. Or I am just
> >>>> confused - these 2 levels of ref counting is really not nice.
> >>>>
> >>>> Anyway, I have just noticed that __mem_cgroup_try_charge does
> >>>> VM_BUG_ON(css_is_removed(&memcg->css)) on a given memcg so you should
> >>>> keep css ref count up as well.
> >>>>

> >>>>
> >>>> IIRC, css_get will prevent the cgroup directory from being removed.
> >>>> Because some allocations are expected to outlive the cgroup, we
> >>>> specifically don't want that.
> >>>
> >>> Yes, but how do you guarantee that the above VM_BUG_ON doesn't trigger?
> >>> Task could have been moved to another group between mem_cgroup_from_task
> >>> and mem_cgroup_get, no?
> >>>
> >>
> >> Ok, after reading this again (and again), you seem to be right. It
> >> concerns me, however, that simply getting the css would lead us to a
> >> double get/put pair, since try_charge will have to do it anyway.
> >
> > That happens only for !*ptr case and you provide a memcg here, don't
> > you.
> >
>
>   if (*ptr) { /* css should be a valid one */
>   memcg = *ptr;
>   VM_BUG_ON(css_is_removed(&memcg->css));
>   if (mem_cgroup_is_root(memcg))
>   goto done;
>   if (consume_stock(memcg, nr_pages))
>   goto done;
>   css_get(&memcg->css);
>
>
> The way I read this, this will still issue a css_get here, unless
> consume_stock suceeds (assuming non-root)
>
> So we'd still have to have a wrapping get/put pair outside the charge.

That is correct but it assumes that the css is valid so somebody upwards
made sure css will not go away. This would suggest css_get is not
necessary here but I guess the primary intention here is to make the
code easier so that we do not have to check whether we took css
reference on the return path.
--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v3 06/13] memcg: kmem controller infrastructure
Posted by Glauber Costa on Mon, 01 Oct 2012 12:04:22 GMT
View Forum Message <> Reply to Message

On 10/01/2012 03:58 PM, Michal Hocko wrote:

> On Mon 01-10-12 15:51:20, Glauber Costa wrote:
>> On 10/01/2012 03:51 PM, Michal Hocko wrote:
>>> On Mon 01-10-12 14:09:09, Glauber Costa wrote:
>>>> On 10/01/2012 01:48 PM, Michal Hocko wrote:
>>>>> On Fri 28-09-12 15:34:19, Glauber Costa wrote:
>>>>>> On 09/27/2012 05:44 PM, Michal Hocko wrote:
>>>>>>>> the reference count aquired by mem_cgroup_get will still prevent the
>>>>>>>> memcg from going away, no?
>>>>>>> Yes but you are outside of the rcu now and we usually do css_get before
>>>>>>> we rcu_unlock. mem_cgroup_get just makes sure the group doesn't get
>>>>>>> deallocated but it could be gone before you call it. Or I am just
>>>>>>> confused - these 2 levels of ref counting is really not nice.
>>>>>>>
>>>>>>> Anyway, I have just noticed that __mem_cgroup_try_charge does
>>>>>>> VM_BUG_ON(css_is_removed(&memcg->css)) on a given memcg so you should
>>>>>>> keep css ref count up as well.
>>>>>>>
>>>>>>
>>>>>> IIRC, css_get will prevent the cgroup directory from being removed.
>>>>>> Because some allocations are expected to outlive the cgroup, we
>>>>>> specifically don't want that.
>>>>>
>>>>> Yes, but how do you guarantee that the above VM_BUG_ON doesn't trigger?
>>>>> Task could have been moved to another group between mem_cgroup_from_task
>>>>> and mem_cgroup_get, no?
>>>>>
>>>>
>>>> Ok, after reading this again (and again), you seem to be right. It
>>>> concerns me, however, that simply getting the css would lead us to a
>>>> double get/put pair, since try_charge will have to do it anyway.
>>>
>>> That happens only for !*ptr case and you provide a memcg here, don't
>>> you.
>>>
>>
>>	if (*ptr) { /* css should be a valid one */
>>		memcg = *ptr;
>>		VM_BUG_ON(css_is_removed(&memcg->css));
>>		if (mem_cgroup_is_root(memcg))
>>			goto done;
>>		if (consume_stock(memcg, nr_pages))
>>			goto done;
>>		css_get(&memcg->css);
>>
>>
>> The way I read this, this will still issue a css_get here, unless
>> consume_stock suceeds (assuming non-root)
>>

>> So we'd still have to have a wrapping get/put pair outside the charge.
>
> That is correct but it assumes that the css is valid so somebody upwards
> made sure css will not go away. This would suggest css_get is not
> necessary here but I guess the primary intention here is to make the
> code easier so that we do not have to check whether we took css
> reference on the return path.
>
In any case, umem would also suffer from double reference, so I'm fine
taking it here as well, since a solution for that is orthogonal.

I still need mem_cgroup_get() to make sure the data structure stays
around, but we only need to do it once at first charge.

Subject: Re: [PATCH v3 09/13] memcg: kmem accounting lifecycle management
Posted by Michal Hocko on Mon, 01 Oct 2012 12:15:53 GMT
View Forum Message <> Reply to Message

Based on the previous discussions I guess this one will get reworked,
right?

On Tue 18-09-12 18:04:06, Glauber Costa wrote:
> 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.
>
> -> /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
> downwards and make sure that we store the information about the parent
> being limited in kmem_accounted.
>
> We do the reverse operation when a formerly limited cgroup becomes
> unlimited.
>
> Since kmem charges may outlive the cgroup existance, we need to be extra

> careful to guarantee the memcg object will stay around for as long as
> needed. Up to now, we were using a mem_cgroup_get()/put() pair in charge
> and uncharge operations.
>
> Although this guarantees that the object will be around until the last
> call to unchage, this means an atomic update in every charge. We can do
> better than that if we only issue get() in the first charge, and then
> put() when the last charge finally goes away.
>
> [v3: merged all lifecycle related patches in one]
>
> Signed-off-by: Glauber Costa <glommer@parallels.com>
> CC: 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 | 123 ++-----
> 1 file changed, 112 insertions(+), 11 deletions(-)
>
> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
> index 0f36a01..720e4bb 100644
> --- a/mm/memcontrol.c
> +++ b/mm/memcontrol.c
> @@ -287,7 +287,8 @@ 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;
> @@ -340,6 +341,43 @@ struct mem_cgroup {
> #endif
> };
>
> +enum {
> + KMEM_ACCOUNTED_ACTIVE = 0, /* accounted by this cgroup itself */
> + KMEM_ACCOUNTED_PARENT, /* one of its parents is active */
> + KMEM_ACCOUNTED_DEAD, /* dead memcg, pending kmem charges */
> +};
> +
> +/* bits 0 and 1 */
> +#define KMEM_ACCOUNTED_MASK 0x3

> +
> +#ifdef CONFIG_MEMCG_KMEM
> +static bool memcg_kmem_set_active(struct mem_cgroup *memcg)
> +{
> + return !test_and_set_bit(KMEM_ACCOUNTED_ACTIVE, &memcg->kmem_accounted);
> +}
> +
> +static bool memcg_kmem_is_accounted(struct mem_cgroup *memcg)
> +{
> + return test_bit(KMEM_ACCOUNTED_ACTIVE, &memcg->kmem_accounted);
> +}
> +
> +static void memcg_kmem_set_active_parent(struct mem_cgroup *memcg)
> +{
> + set_bit(KMEM_ACCOUNTED_PARENT, &memcg->kmem_accounted);
> +}
> +
> +static void memcg_kmem_mark_dead(struct mem_cgroup *memcg)
> +{
> + if (test_bit(KMEM_ACCOUNTED_ACTIVE, &memcg->kmem_accounted))
> + set_bit(KMEM_ACCOUNTED_DEAD, &memcg->kmem_accounted);
> +}
> +
> +static bool memcg_kmem_dead(struct mem_cgroup *memcg)
> +{
> + return test_and_clear_bit(KMEM_ACCOUNTED_DEAD, &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
> @@ -491,7 +529,7 @@ EXPORT_SYMBOL(tcp_proto_cgroup);
> static inline bool memcg_can_account_kmem(struct mem_cgroup *memcg)
> {
> return !mem_cgroup_disabled() && !mem_cgroup_is_root(memcg) &&
> - memcg->kmem_accounted;
> + (memcg->kmem_accounted & (KMEM_ACCOUNTED_MASK));
> }
>
> /*
> @@ -524,13 +562,9 @@ __memcg_kmem_newpage_charge(gfp_t gfp, struct mem_cgroup
**_memcg, int order)
> if (!memcg_can_account_kmem(memcg))
> return true;
>
> - mem_cgroup_get(memcg);
> -

>   ret = memcg_charge_kmem(memcg, gfp, PAGE_SIZE << order) == 0;
>   if (ret)
>   *_memcg = memcg;
> - else
> - mem_cgroup_put(memcg);
>
>   return ret;
> }
> @@ -589,7 +623,6 @@ void __memcg_kmem_uncharge_page(struct page *page, int order)
>
>   WARN_ON(mem_cgroup_is_root(memcg));
>   memcg_uncharge_kmem(memcg, PAGE_SIZE << order);
> - mem_cgroup_put(memcg);
> }
> #endif /* CONFIG_MEMCG_KMEM */
>
> @@ -4077,6 +4110,40 @@ 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
> + struct mem_cgroup *iter;
> +
> + /*
> + * When we are doing hierarchical accounting, with an hierarchy like
> + * A/B/C, we need to start accounting kernel memory all the way up to C
> + * in case A start being accounted.
> + *
> + * So when we the cgroup first gets to be unlimited, we walk all the
> + * children of the current memcg and enable kmem accounting for them.
> + * Note that a separate bit is used there to indicate that the
> + * accounting happens due to the parent being accounted.
> + *
> + * note that memcg_kmem_set_active is a test-and-set routine, so we only
> + * arrive here once (since we never disable it)
> + */
> + mutex_lock(&set_limit_mutex);
> + if ((val != RESOURCE_MAX) && memcg_kmem_set_active(memcg)) {
> +
> + mem_cgroup_get(memcg);
> +
> + for_each_mem_cgroup_tree(iter, memcg) {
> +   if (iter == memcg)
> +   continue;

> +   memcg_kmem_set_active_parent(iter);
> + }
> + }
> + mutex_unlock(&set_limit_mutex);
> +#endif
> +}
> +
> /*
>   * The user of this function is...
>   * RES_LIMIT.
> @@ -4115,9 +4182,7 @@ static int mem_cgroup_write(struct cgroup *cont, struct cftype *cft,
>   if (ret)
>   break;
>
> -   /* For simplicity, we won't allow this to be disabled */
> -   if (!memcg->kmem_accounted && val != RESOURCE_MAX)
> -   memcg->kmem_accounted = true;
> +   memcg_update_kmem_limit(memcg, val);
>   } else
>   return -EINVAL;
>   break;
> @@ -4791,6 +4856,20 @@ static int memcg_init_kmem(struct mem_cgroup *memcg, struct
cgroup_subsys *ss)
> static void kmem_cgroup_destroy(struct mem_cgroup *memcg)
> {
>   mem_cgroup_sockets_destroy(memcg);
> +
> + memcg_kmem_mark_dead(memcg);
> +
> + if (res_counter_read_u64(&memcg->kmem, RES_USAGE) != 0)
> + return;
> +
> + /*
> + * Charges already down to 0, undo mem_cgroup_get() done in the charge
> + * path here, being careful not to race with memcg_uncharge_kmem: it is
> + * possible that the charges went down to 0 between mark_dead and the
> + * res_counter read, so in that case, we don't need the put
> + */
> + if (memcg_kmem_dead(memcg))
> + mem_cgroup_put(memcg);
> }
> #else
> static int memcg_init_kmem(struct mem_cgroup *memcg, struct cgroup_subsys *ss)
> @@ -5148,6 +5227,8 @@ mem_cgroup_create(struct cgroup *cont)
>   }
>
>   if (parent && parent->use_hierarchy) {
> + struct mem_cgroup __maybe_unused *p;

```
> +
>    res_counter_init(&memcg->res, &parent->res);
>    res_counter_init(&memcg->memsw, &parent->memsw);
>    res_counter_init(&memcg->kmem, &parent->kmem);
> @@ -5158,6 +5239,20 @@ mem_cgroup_create(struct cgroup *cont)
>     * mem_cgroup(see mem_cgroup_put).
>     */
>    mem_cgroup_get(parent);
> +#ifdef CONFIG_MEMCG_KMEM
> +  /*
> +   * In case a parent is already limited when we create this, we
> +   * need him to propagate it now so we become limited as well.
> +   */
> +  mutex_lock(&set_limit_mutex);
> +  for (p = parent; p != NULL; p = parent_mem_cgroup(p)) {
> +   if (memcg_kmem_is_accounted(p)) {
> +    memcg_kmem_set_active_parent(memcg);
> +    break;
> +   }
> +  }
> +  mutex_unlock(&set_limit_mutex);
> +#endif
>   } else {
>    res_counter_init(&memcg->res, NULL);
>    res_counter_init(&memcg->memsw, NULL);
> @@ -5871,9 +5966,15 @@ void memcg_uncharge_kmem(struct mem_cgroup *memcg, u64
size)
>   if (!memcg)
>    return;
>
> - res_counter_uncharge(&memcg->kmem, size);
>   res_counter_uncharge(&memcg->res, size);
>   if (do_swap_account)
>    res_counter_uncharge(&memcg->memsw, size);
> +
> + /* Not down to 0 */
> + if (res_counter_uncharge(&memcg->kmem, size))
> +  return;
> +
> + if (memcg_kmem_dead(memcg))
> +  mem_cgroup_put(memcg);
>  }
>  #endif /* CONFIG_MEMCG_KMEM */
> --
> 1.7.11.4
>
> --
> 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 v3 10/13] memcg: use static branches when code not in use
Posted by Michal Hocko on Mon, 01 Oct 2012 12:25:17 GMT
View Forum Message <> Reply to Message

On Tue 18-09-12 18:04:07, Glauber Costa wrote:
[...]
> include/linux/memcontrol.h | 4 +++-
> mm/memcontrol.c | 26 ++++++++++++++++++++++++--
> 2 files changed, 27 insertions(+), 3 deletions(-)
>
> diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
> index 82ede9a..4ec9fd5 100644
> --- a/include/linux/memcontrol.h
> +++ b/include/linux/memcontrol.h
> @@ -22,6 +22,7 @@
> #include <linux/cgroup.h>
> #include <linux/vm_event_item.h>
> #include <linux/hardirq.h>
> +#include <linux/jump_label.h>
>
> struct mem_cgroup;
> struct page_cgroup;
> @@ -401,9 +402,10 @@ struct sock;
> void sock_update_memcg(struct sock *sk);
> void sock_release_memcg(struct sock *sk);
>
> +extern struct static_key memcg_kmem_enabled_key;
> static inline bool memcg_kmem_enabled(void)
> {
> - return true;
> + return static_key_false(&memcg_kmem_enabled_key);
> }
>
> extern bool __memcg_kmem_newpage_charge(gfp_t gfp, struct mem_cgroup **memcg,
> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
> index 720e4bb..aada601 100644
> --- a/mm/memcontrol.c
> +++ b/mm/memcontrol.c
> @@ -467,6 +467,8 @@ 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;
> +
> static bool mem_cgroup_is_root(struct mem_cgroup *memcg);
> static int memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t gfp, u64 size);
> static void memcg_uncharge_kmem(struct mem_cgroup *memcg, u64 size);
> @@ -624,6 +626,16 @@ void __memcg_kmem_uncharge_page(struct page *page, int order)
>   WARN_ON(mem_cgroup_is_root(memcg));
>   memcg_uncharge_kmem(memcg, PAGE_SIZE << order);
> }
> +
> +static void disarm_kmem_keys(struct mem_cgroup *memcg)
> +{
> + if (memcg_kmem_is_accounted(memcg))
> + 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)
> @@ -639,6 +651,12 @@ 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 *
> @@ -4131,7 +4149,11 @@ static void memcg_update_kmem_limit(struct mem_cgroup
*memcg, u64 val)
>   */
>   mutex_lock(&set_limit_mutex);
>   if ((val != RESOURCE_MAX) && memcg_kmem_set_active(memcg)) {
> -
> + /*
> +   * Once the static branch is enabled it will only be
> +   * disabled when the last reference to memcg is gone.
> +   */

> + static_key_slow_inc(&memcg_kmem_enabled_key);

I guess the reason why we do not need to inc also for children is that
we do not inherit kmem_accounted, right?

>   mem_cgroup_get(memcg);
>
>   for_each_mem_cgroup_tree(iter, memcg) {
> @@ -5066,7 +5088,7 @@ 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
> --
> 1.7.11.4
>
> --
> 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 v3 10/13] memcg: use static branches when code not in use
Posted by Glauber Costa on Mon, 01 Oct 2012 12:27:27 GMT
View Forum Message <> Reply to Message

On 10/01/2012 04:25 PM, Michal Hocko wrote:
> On Tue 18-09-12 18:04:07, Glauber Costa wrote:
> [...]
>> include/linux/memcontrol.h | 4 +++-
>> mm/memcontrol.c | 26 ++++++++++++++++++++++++--
>> 2 files changed, 27 insertions(+), 3 deletions(-)
>>
>> diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
>> index 82ede9a..4ec9fd5 100644
>> --- a/include/linux/memcontrol.h
>> +++ b/include/linux/memcontrol.h
>> @@ -22,6 +22,7 @@
>> #include <linux/cgroup.h>
>> #include <linux/vm_event_item.h>

>> #include <linux/hardirq.h>
>> +#include <linux/jump_label.h>
>>
>> struct mem_cgroup;
>> struct page_cgroup;
>> @@ -401,9 +402,10 @@ struct sock;
>> void sock_update_memcg(struct sock *sk);
>> void sock_release_memcg(struct sock *sk);
>>
>> +extern struct static_key memcg_kmem_enabled_key;
>> static inline bool memcg_kmem_enabled(void)
>> {
>> - return true;
>> + return static_key_false(&memcg_kmem_enabled_key);
>> }
>>
>> extern bool __memcg_kmem_newpage_charge(gfp_t gfp, struct mem_cgroup **memcg,
>> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
>> index 720e4bb..aada601 100644
>> --- a/mm/memcontrol.c
>> +++ b/mm/memcontrol.c
>> @@ -467,6 +467,8 @@ 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;
>> +
>> static bool mem_cgroup_is_root(struct mem_cgroup *memcg);
>> static int memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t gfp, u64 size);
>> static void memcg_uncharge_kmem(struct mem_cgroup *memcg, u64 size);
>> @@ -624,6 +626,16 @@ void __memcg_kmem_uncharge_page(struct page *page, int order)
>>   WARN_ON(mem_cgroup_is_root(memcg));
>>   memcg_uncharge_kmem(memcg, PAGE_SIZE << order);
>> }
>> +
>> +static void disarm_kmem_keys(struct mem_cgroup *memcg)
>> +{
>> + if (memcg_kmem_is_accounted(memcg))
>> + 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)

>> @@ -639,6 +651,12 @@ 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 *
>> @@ -4131,7 +4149,11 @@ static void memcg_update_kmem_limit(struct mem_cgroup
*memcg, u64 val)
>>   */
>>   mutex_lock(&set_limit_mutex);
>>   if ((val != RESOURCE_MAX) && memcg_kmem_set_active(memcg)) {
>> -
>> + /*
>> +   * Once the static branch is enabled it will only be
>> +   * disabled when the last reference to memcg is gone.
>> +   */
>> + static_key_slow_inc(&memcg_kmem_enabled_key);
>
> I guess the reason why we do not need to inc also for children is that
> we do not inherit kmem_accounted, right?
>

Yes, but I of course changed that in the upcoming version of the patch.
We now inherit the value everytime, and the static branches are updated
accordingly.

Subject: Re: [PATCH v3 09/13] memcg: kmem accounting lifecycle management
Posted by Glauber Costa on Mon, 01 Oct 2012 12:29:11 GMT
View Forum Message <> Reply to Message

On 10/01/2012 04:15 PM, Michal Hocko wrote:
> Based on the previous discussions I guess this one will get reworked,
> right?
>

Yes, but most of it stayed. The hierarchy part is gone, but because we
will still have kmem pages floating around (potentially), I am still
using the mark_dead() trick with the corresponding get when kmem_accounted.

Subject: Re: [PATCH v3 11/13] memcg: allow a memcg with kmem charges to be destructed.
Posted by Michal Hocko on Mon, 01 Oct 2012 12:30:36 GMT
View Forum Message <> Reply to Message

On Tue 18-09-12 18:04:08, 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.
>
> 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>

Looks good.
Reviewed-by: Michal Hocko <mhocko@suse.cz>

> ---
> mm/memcontrol.c | 17 ++++++++++++++++-
> 1 file changed, 16 insertions(+), 1 deletion(-)
>
> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
> index aada601..b05ecac 100644
> --- a/mm/memcontrol.c
> +++ b/mm/memcontrol.c
> @@ -631,6 +631,11 @@ static void disarm_kmem_keys(struct mem_cgroup *memcg)
> {
>   if (memcg_kmem_is_accounted(memcg))
>   static_key_slow_dec(&memcg_kmem_enabled_key);
> + /*
> + * This check can't live in kmem destruction function,
> + * since the charges will outlive the cgroup
> + */
> + WARN_ON(res_counter_read_u64(&memcg->kmem, RES_USAGE) != 0);
> }
> #else
> static void disarm_kmem_keys(struct mem_cgroup *memcg)
> @@ -3933,6 +3938,7 @@ 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);
>
> @@ -3966,8 +3972,17 @@ 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;
> --
> 1.7.11.4
>
> --
> 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 v3 09/13] memcg: kmem accounting lifecycle management
Posted by Michal Hocko on Mon, 01 Oct 2012 12:36:54 GMT
View Forum Message <> Reply to Message

On Mon 01-10-12 16:29:11, Glauber Costa wrote:
> On 10/01/2012 04:15 PM, Michal Hocko wrote:
> > Based on the previous discussions I guess this one will get reworked,
> > right?
> >

>
> Yes, but most of it stayed. The hierarchy part is gone, but because we
> will still have kmem pages floating around (potentially), I am still
> using the mark_dead() trick with the corresponding get when kmem_accounted.

Is it OK if I hold on with the review of this one until the next
version?
--
Michal Hocko
SUSE Labs

Subject: Re: [PATCH v3 09/13] memcg: kmem accounting lifecycle management
Posted by Glauber Costa on Mon, 01 Oct 2012 12:43:05 GMT
View Forum Message <> Reply to Message

On 10/01/2012 04:36 PM, Michal Hocko wrote:
> On Mon 01-10-12 16:29:11, Glauber Costa wrote:
>> On 10/01/2012 04:15 PM, Michal Hocko wrote:
>>> Based on the previous discussions I guess this one will get reworked,
>>> right?
>>>
>>
>> Yes, but most of it stayed. The hierarchy part is gone, but because we
>> will still have kmem pages floating around (potentially), I am still
>> using the mark_dead() trick with the corresponding get when kmem_accounted.
>
> Is it OK if I hold on with the review of this one until the next
> version?
>
Of course.

I haven't sent it yet because I also received a lot more feedback for
the slab part (which is expected), and I want to get a least part of
that going before I send it again.

Subject: Re: [PATCH v3 13/13] protect architectures where THREAD_SIZE >= PAGE_SIZE against fork bombs
Posted by Michal Hocko on Mon, 01 Oct 2012 13:17:09 GMT
View Forum Message <> Reply to Message

On Tue 18-09-12 18:04:10, 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 :(){ :|:& };:

OK. Although I was complaining that this is not the full truth the last
time, I do not insist on gravy details about the slaughter this will
cause to the rest of the group and that who-ever could fork in the group
can easily DOS the whole hierarchy. It has some interesting side effects
as well but let's keep this to a careful reader ;)

The patch, as is, is still useful and an improvement because it reduces
the impact.

>
> Signed-off-by: Glauber Costa <glommer@parallels.com>
> Acked-by: Frederic Weisbecker <fweisbec@redhat.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>

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 0ff2bf7..897e89c 100644
> --- a/kernel/fork.c
> +++ b/kernel/fork.c
> @@ -146,7 +146,7 @@ void __weak arch_release_thread_info(struct thread_info *ti)
> static struct thread_info *alloc_thread_info_node(struct task_struct *tsk,
> int node)
> {
> - struct page *page = alloc_pages_node(node, THREADINFO_GFP,
> + struct page *page = alloc_pages_node(node, THREADINFO_GFP_ACCOUNTED,
> THREAD_SIZE_ORDER);
>
>   return page ? page_address(page) : NULL;
> @@ -154,7 +154,7 @@ static struct thread_info *alloc_thread_info_node(struct task_struct
*tsk,
>
> static inline void free_thread_info(struct thread_info *ti)
> {
> - free_pages((unsigned long)ti, THREAD_SIZE_ORDER);
> + free_accounted_pages((unsigned long)ti, THREAD_SIZE_ORDER);
> }
> # else
> static struct kmem_cache *thread_info_cache;
> --
> 1.7.11.4
>
> --
> 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 v3 12/13] execute the whole memcg freeing in rcu callback
Posted by Michal Hocko on Mon, 01 Oct 2012 13:27:11 GMT
View Forum Message <> Reply to Message

On Tue 18-09-12 18:04:09, Glauber Costa wrote:
> A lot of the initialization we do in mem_cgroup_create() is done with softirqs

> enabled. This include grabbing a css id, which holds &ss->id_lock->rlock, and
> the per-zone trees, which holds rtpz->lock->rlock. All of those signal to the
> lockdep mechanism that those locks can be used in SOFTIRQ-ON-W context. This
> means that the freeing of memcg structure must happen in a compatible context,
> otherwise we'll get a deadlock.

Maybe I am missing something obvious but why cannot we simply disble
(soft)irqs in mem_cgroup_create rather than make the free path much more
complicated. It really feels strange to defer everything (e.g. soft
reclaim tree cleanup which should be a no-op at the time because there
shouldn't be any user pages in the group).

> The reference counting mechanism we use allows the memcg structure to be freed
> later and outlive the actual memcg destruction from the filesystem. However, we
> have little, if any, means to guarantee in which context the last memcg_put
> will happen. The best we can do is test it and try to make sure no invalid
> context releases are happening. But as we add more code to memcg, the possible
> interactions grow in number and expose more ways to get context conflicts.
>
> We already moved a part of the freeing to a worker thread to be context-safe
> for the static branches disabling. I see no reason not to do it for the whole
> freeing action. I consider this to be the safe choice.

>
> Signed-off-by: Glauber Costa <glommer@parallels.com>
> Tested-by: Greg Thelen <gthelen@google.com>
> CC: KAMEZAWA Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
> CC: Michal Hocko <mhocko@suse.cz>
> CC: Johannes Weiner <hannes@cmpxchg.org>
> ---
> mm/memcontrol.c | 66 ++++++++++++++++++++++++++++++++----------------------
> 1 file changed, 34 insertions(+), 32 deletions(-)
>
> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
> index b05ecac..74654f0 100644
> --- a/mm/memcontrol.c
> +++ b/mm/memcontrol.c
> @@ -5082,16 +5082,29 @@ out_free:
> }
>
> /*
> - * Helpers for freeing a kmalloc()ed/vzalloc()ed mem_cgroup by RCU,
> - * but in process context. The work_freeing structure is overlaid
> - * on the rcu_freeing structure, which itself is overlaid on memsw.
> + * At destroying mem_cgroup, references from swap_cgroup can remain.
> + * (scanning all at force_empty is too costly...)
> + *

> + * Instead of clearing all references at force_empty, we remember
> + * the number of reference from swap_cgroup and free mem_cgroup when
> + * it goes down to 0.
> + *
> + * Removal of cgroup itself succeeds regardless of refs from swap.
>   */
> -static void free_work(struct work_struct *work)
> +
> +static void __mem_cgroup_free(struct mem_cgroup *memcg)
> {
> - struct mem_cgroup *memcg;
> + int node;
>   int size = sizeof(struct mem_cgroup);
>
> - memcg = container_of(work, struct mem_cgroup, work_freeing);
> + mem_cgroup_remove_from_trees(memcg);
> + free_css_id(&mem_cgroup_subsys, &memcg->css);
> +
> + for_each_node(node)
> + free_mem_cgroup_per_zone_info(memcg, node);
> +
> + free_percpu(memcg->stat);
> +
>   /*
>   * We need to make sure that (at least for now), the jump label
>   * destruction code runs outside of the cgroup lock. This is because
> @@ -5110,38 +5123,27 @@ static void free_work(struct work_struct *work)
>   vfree(memcg);
> }
>
> -static void free_rcu(struct rcu_head *rcu_head)
> -{
> - struct mem_cgroup *memcg;
> -
> - memcg = container_of(rcu_head, struct mem_cgroup, rcu_freeing);
> - INIT_WORK(&memcg->work_freeing, free_work);
> - schedule_work(&memcg->work_freeing);
> -}
>
> /*
> - * At destroying mem_cgroup, references from swap_cgroup can remain.
> - * (scanning all at force_empty is too costly...)
> - *
> - * Instead of clearing all references at force_empty, we remember
> - * the number of reference from swap_cgroup and free mem_cgroup when
> - * it goes down to 0.
> - *
> - * Removal of cgroup itself succeeds regardless of refs from swap.

> + * Helpers for freeing a kmalloc()ed/vzalloc()ed mem_cgroup by RCU,
> + * but in process context. The work_freeing structure is overlaid
> + * on the rcu_freeing structure, which itself is overlaid on memsw.
>   */
> -
> -static void __mem_cgroup_free(struct mem_cgroup *memcg)
> +static void free_work(struct work_struct *work)
> {
> - int node;
> + struct mem_cgroup *memcg;
>
> - mem_cgroup_remove_from_trees(memcg);
> - free_css_id(&mem_cgroup_subsys, &memcg->css);
> + memcg = container_of(work, struct mem_cgroup, work_freeing);
> + __mem_cgroup_free(memcg);
> +}
>
> - for_each_node(node)
> - free_mem_cgroup_per_zone_info(memcg, node);
> +static void free_rcu(struct rcu_head *rcu_head)
> +{
> + struct mem_cgroup *memcg;
>
> - free_percpu(memcg->stat);
> - call_rcu(&memcg->rcu_freeing, free_rcu);
> + memcg = container_of(rcu_head, struct mem_cgroup, rcu_freeing);
> + INIT_WORK(&memcg->work_freeing, free_work);
> + schedule_work(&memcg->work_freeing);
> }
>
> static void mem_cgroup_get(struct mem_cgroup *memcg)
> @@ -5153,7 +5155,7 @@ static void __mem_cgroup_put(struct mem_cgroup *memcg, int
count)
> {
>   if (atomic_sub_and_test(count, &memcg->refcnt)) {
>   struct mem_cgroup *parent = parent_mem_cgroup(memcg);
> - __mem_cgroup_free(memcg);
> + call_rcu(&memcg->rcu_freeing, free_rcu);
>   if (parent)
>   mem_cgroup_put(parent);
>   }
> --
> 1.7.11.4
>
> --
> 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 v3 01/13] memcg: Make it possible to use the stock for more than one page.
Posted by Johannes Weiner on Mon, 01 Oct 2012 18:48:12 GMT
View Forum Message <> Reply to Message

On Tue, Sep 18, 2012 at 06:03:58PM +0400, 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>

Independent of how the per-subtree enable-through-setting-limit
discussion pans out, we're going to need the charge cache, so:

Acked-by: Johannes Weiner <hannes@cmpxchg.org>

Subject: Re: [PATCH v3 03/13] memcg: change defines to an enum
Posted by Johannes Weiner on Mon, 01 Oct 2012 19:06:52 GMT
View Forum Message <> Reply to Message

On Tue, Sep 18, 2012 at 06:04:00PM +0400, 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: Johannes Weiner <hannes@cmpxchg.org>
> Acked-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
> Acked-by: Michal Hocko <mhocko@suse.cz>

Should probably be the first in the series to get the cleanups out of
the way :-)

Acked-by: Johannes Weiner <hannes@cmpxchg.org>

Subject: Re: [PATCH v3 05/13] Add a __GFP_KMEMCG flag
Posted by Johannes Weiner on Mon, 01 Oct 2012 19:09:05 GMT
View Forum Message <> Reply to Message

On Tue, Sep 18, 2012 at 06:04:02PM +0400, 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: 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>
> Acked-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>

With the feedback from Christoph and Mel incorporated:

Acked-by: Johannes Weiner <hannes@cmpxchg.org>

Subject: Re: [PATCH v3 03/13] memcg: change defines to an enum
Posted by Glauber Costa on Tue, 02 Oct 2012 09:10:23 GMT
View Forum Message <> Reply to Message

On 10/01/2012 11:06 PM, Johannes Weiner wrote:
> On Tue, Sep 18, 2012 at 06:04:00PM +0400, 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: Johannes Weiner <hannes@cmpxchg.org>
>> Acked-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
>> Acked-by: Michal Hocko <mhocko@suse.cz>
>
> Should probably be the first in the series to get the cleanups out of
> the way :-)
>
> Acked-by: Johannes Weiner <hannes@cmpxchg.org>
>
If you guys want to merge this separately, be my guest =)

Subject: Re: [PATCH v3 06/13] memcg: kmem controller infrastructure
Posted by Tejun Heo on Wed, 03 Oct 2012 22:11:51 GMT
View Forum Message <> Reply to Message

Hello, Glauber.

Sorry about late replies. I'be been traveling for the Korean
thanksgiving holidays.

On Mon, Oct 01, 2012 at 12:28:28PM +0400, Glauber Costa wrote:
> > That synchronous ref draining is going away. Maybe we can do that
> > before kmemcg? Michal, do you have some timeframe on mind?
>
> Since you said yourself in other points in this thread that you are fine
> with some page references outliving the cgroup in the case of slab, this
> is a situation that comes with the code, not a situation that was
> incidentally there, and we're making use of.

Hmmm? Not sure what you're trying to say but I wanted to say that
this should be okay once the scheduled memcg pre_destroy change
happens and nudge Michal once more.

Thanks.

--
tejun

Subject: Re: [PATCH v3 12/13] execute the whole memcg freeing in rcu callback
Posted by Glauber Costa on Thu, 04 Oct 2012 10:53:13 GMT
View Forum Message <> Reply to Message

On 10/01/2012 05:27 PM, Michal Hocko wrote:
> On Tue 18-09-12 18:04:09, Glauber Costa wrote:
>> A lot of the initialization we do in mem_cgroup_create() is done with softirqs

>> enabled. This include grabbing a css id, which holds &ss->id_lock->rlock, and
>> the per-zone trees, which holds rtpz->lock->rlock. All of those signal to the
>> lockdep mechanism that those locks can be used in SOFTIRQ-ON-W context. This
>> means that the freeing of memcg structure must happen in a compatible context,
>> otherwise we'll get a deadlock.
>
> Maybe I am missing something obvious but why cannot we simply disble
> (soft)irqs in mem_cgroup_create rather than make the free path much more
> complicated. It really feels strange to defer everything (e.g. soft
> reclaim tree cleanup which should be a no-op at the time because there
> shouldn't be any user pages in the group).
>

Ok.

I was just able to come back to this today - I was mostly working on the slab feedback over the past few days. I will answer yours and Tejun's concerns at once:

Here is the situation: the backtrace I get is this one:

```
[  124.956725] ================================
[  124.957217] [ INFO: inconsistent lock state ]
[  124.957217] 3.5.0+ #99 Not tainted
[  124.957217] --------------------------------
[  124.957217] inconsistent {SOFTIRQ-ON-W} -> {IN-SOFTIRQ-W} usage.
[  124.957217] ksoftirqd/0/3 [HC0[0]:SC1[1]:HE1:SE0] takes:
[  124.957217]  (&(&ss->id_lock)->rlock){+.?...}, at:
[<ffffffff810aa7b2>] spin_lock+0x9/0xb
[  124.957217] {SOFTIRQ-ON-W} state was registered at:
[  124.957217]   [<ffffffff810996ed>] __lock_acquire+0x31f/0xd68
[  124.957217]   [<ffffffff8109a660>] lock_acquire+0x108/0x15c
[  124.957217]   [<ffffffff81534ec4>] _raw_spin_lock+0x40/0x4f
[  124.957217]   [<ffffffff810aa7b2>] spin_lock+0x9/0xb
[  124.957217]   [<ffffffff810ad00e>] get_new_cssid+0x69/0xf3
[  124.957217]   [<ffffffff810ad0da>] cgroup_init_idr+0x42/0x60
[  124.957217]   [<ffffffff81b20e04>] cgroup_init+0x50/0x100
[  124.957217]   [<ffffffff81b05b9b>] start_kernel+0x3b9/0x3ee
[  124.957217]   [<ffffffff81b052d6>] x86_64_start_reservations+0xb1/0xb5
[  124.957217]   [<ffffffff81b053d8>] x86_64_start_kernel+0xfe/0x10b
```

So what we learn from it, is: we are acquiring a specific lock (the css id one) from softirq context. It was previously taken in a softirq-enabled context, that seems to be coming directly from get_new_cssid.

Tejun correctly pointed out that we should never acquire that lock from

a softirq context, in which he is right.

But the situation changes slightly with kmem. Now, the following excerpt
of a backtrace is possible:

[48.602775] [<ffffffff81103095>] free_accounted_pages+0x47/0x4c
[48.602775] [<ffffffff81047f90>] free_task+0x31/0x5c
[48.602775] [<ffffffff8104807d>] __put_task_struct+0xc2/0xdb
[48.602775] [<ffffffff8104dfc7>] put_task_struct+0x1e/0x22
[48.602775] [<ffffffff8104e144>] delayed_put_task_struct+0x7a/0x98
[48.602775] [<ffffffff810cf0e5>] __rcu_process_callbacks+0x269/0x3df
[48.602775] [<ffffffff810cf28c>] rcu_process_callbacks+0x31/0x5b
[48.602775] [<ffffffff8105266d>] __do_softirq+0x122/0x277

So as you can see, free_accounted_pages (that will trigger a memcg_put()
-> mem_cgroup_free()) can now be called from softirq context, which is,
an rcu callback (and I just realized I wrote the exact opposite in the
subj line: man, I really suck at that!!)
As a matter of fact, we could not move to our rcu callback as well:

we need to move it to a worker thread with the rest.

We already have a worker thread: he reason we have it is not
static_branches: The reason is vfree(), that will BUG_ON(in_interrupt())
and could not be called from rcu callback as well. We moved static
branches in there as well for a similar problem, but haven't introduced it.

Could we move just part of it to the worker thread? Absolutely yes.
Moving just free_css_id() is enough to make it work. But since it is not
the first context related problem we had, I thought: "to hell with that,
let's move everything and be safe".

I am fine moving free_css_id() only if you would prefer.

Can we disable softirqs when we initialize css_id? Maybe. My machine
seems to boot fine and survive the simple workload that would trigger
that bug if I use irqsave spinlocks instead of normal spinlocks. But
this has to be done from cgroup core: We have no control over css
creation in memcg.

How would you guys like me to handle this ?

Subject: Re: [PATCH v3 12/13] execute the whole memcg freeing in rcu callback
Posted by Glauber Costa on Thu, 04 Oct 2012 14:20:07 GMT
View Forum Message <> Reply to Message

On 10/04/2012 02:53 PM, Glauber Costa wrote:

> On 10/01/2012 05:27 PM, Michal Hocko wrote:
>> On Tue 18-09-12 18:04:09, Glauber Costa wrote:
>>> A lot of the initialization we do in mem_cgroup_create() is done with softirqs
>>> enabled. This include grabbing a css id, which holds &ss->id_lock->rlock, and
>>> the per-zone trees, which holds rtpz->lock->rlock. All of those signal to the
>>> lockdep mechanism that those locks can be used in SOFTIRQ-ON-W context. This
>>> means that the freeing of memcg structure must happen in a compatible context,
>>> otherwise we'll get a deadlock.
>>
>> Maybe I am missing something obvious but why cannot we simply disble
>> (soft)irqs in mem_cgroup_create rather than make the free path much more
>> complicated. It really feels strange to defer everything (e.g. soft
>> reclaim tree cleanup which should be a no-op at the time because there
>> shouldn't be any user pages in the group).
>>
>
> Ok.
>
> I was just able to come back to this today - I was mostly working on the
> slab feedback over the past few days. I will answer yours and Tejun's
> concerns at once:
>
> Here is the situation: the backtrace I get is this one:
>
> [124.956725] ================================
> [124.957217] [INFO: inconsistent lock state]
> [124.957217] 3.5.0+ #99 Not tainted
> [124.957217] --------------------------------
> [124.957217] inconsistent {SOFTIRQ-ON-W} -> {IN-SOFTIRQ-W} usage.
> [124.957217] ksoftirqd/0/3 [HC0[0]:SC1[1]:HE1:SE0] takes:
> [124.957217] (&(&ss->id_lock)->rlock){+.?...}, at:
> [<ffffffff810aa7b2>] spin_lock+0x9/0xb
> [124.957217] {SOFTIRQ-ON-W} state was registered at:
> [124.957217]   [<ffffffff810996ed>] __lock_acquire+0x31f/0xd68
> [124.957217]   [<ffffffff8109a660>] lock_acquire+0x108/0x15c
> [124.957217]   [<ffffffff81534ec4>] _raw_spin_lock+0x40/0x4f
> [124.957217]   [<ffffffff810aa7b2>] spin_lock+0x9/0xb
> [124.957217]   [<ffffffff810ad00e>] get_new_cssid+0x69/0xf3
> [124.957217]   [<ffffffff810ad0da>] cgroup_init_idr+0x42/0x60
> [124.957217]   [<ffffffff81b20e04>] cgroup_init+0x50/0x100
> [124.957217]   [<ffffffff81b05b9b>] start_kernel+0x3b9/0x3ee
> [124.957217]   [<ffffffff81b052d6>] x86_64_start_reservations+0xb1/0xb5
> [124.957217]   [<ffffffff81b053d8>] x86_64_start_kernel+0xfe/0x10b
>
>
> So what we learn from it, is: we are acquiring a specific lock (the css
> id one) from softirq context. It was previously taken in a
> softirq-enabled context, that seems to be coming directly from

> get_new_cssid.
>
> Tejun correctly pointed out that we should never acquire that lock from
> a softirq context, in which he is right.
>
> But the situation changes slightly with kmem. Now, the following excerpt
> of a backtrace is possible:
>
> [48.602775] [<ffffffff81103095>] free_accounted_pages+0x47/0x4c
> [48.602775] [<ffffffff81047f90>] free_task+0x31/0x5c
> [48.602775] [<ffffffff8104807d>] __put_task_struct+0xc2/0xdb
> [48.602775] [<ffffffff8104dfc7>] put_task_struct+0x1e/0x22
> [48.602775] [<ffffffff8104e144>] delayed_put_task_struct+0x7a/0x98
> [48.602775] [<ffffffff810cf0e5>] __rcu_process_callbacks+0x269/0x3df
> [48.602775] [<ffffffff810cf28c>] rcu_process_callbacks+0x31/0x5b
> [48.602775] [<ffffffff8105266d>] __do_softirq+0x122/0x277
>
> So as you can see, free_accounted_pages (that will trigger a memcg_put()
> -> mem_cgroup_free()) can now be called from softirq context, which is,
> an rcu callback (and I just realized I wrote the exact opposite in the
> subj line: man, I really suck at that!!)
> As a matter of fact, we could not move to our rcu callback as well:
>
> we need to move it to a worker thread with the rest.
>
> We already have a worker thread: he reason we have it is not
> static_branches: The reason is vfree(), that will BUG_ON(in_interrupt())
> and could not be called from rcu callback as well. We moved static
> branches in there as well for a similar problem, but haven't introduced it.
>
> Could we move just part of it to the worker thread? Absolutely yes.
> Moving just free_css_id() is enough to make it work. But since it is not
> the first context related problem we had, I thought: "to hell with that,
> let's move everything and be safe".
>
> I am fine moving free_css_id() only if you would prefer.
>
> Can we disable softirqs when we initialize css_id? Maybe. My machine
> seems to boot fine and survive the simple workload that would trigger
> that bug if I use irqsave spinlocks instead of normal spinlocks. But
> this has to be done from cgroup core: We have no control over css
> creation in memcg.
>
> How would you guys like me to handle this ?

One more thing: As I mentioned in the Changelog,
mem_cgroup_remove_exceeded(), called from mem_cgroup_remove_from_trees()
will lead to the same usage pattern.

Subject: Re: [PATCH v3 12/13] execute the whole memcg freeing in rcu callback
Posted by Johannes Weiner on Fri, 05 Oct 2012 15:31:00 GMT
View Forum Message <> Reply to Message

On Thu, Oct 04, 2012 at 02:53:13PM +0400, Glauber Costa wrote:
> On 10/01/2012 05:27 PM, Michal Hocko wrote:
> > On Tue 18-09-12 18:04:09, Glauber Costa wrote:
> >> A lot of the initialization we do in mem_cgroup_create() is done with softirqs
> >> enabled. This include grabbing a css id, which holds &ss->id_lock->rlock, and
> >> the per-zone trees, which holds rtpz->lock->rlock. All of those signal to the
> >> lockdep mechanism that those locks can be used in SOFTIRQ-ON-W context. This
> >> means that the freeing of memcg structure must happen in a compatible context,
> >> otherwise we'll get a deadlock.
> >
> > Maybe I am missing something obvious but why cannot we simply disble
> > (soft)irqs in mem_cgroup_create rather than make the free path much more
> > complicated. It really feels strange to defer everything (e.g. soft
> > reclaim tree cleanup which should be a no-op at the time because there
> > shouldn't be any user pages in the group).
> >
>
> Ok.
>
> I was just able to come back to this today - I was mostly working on the
> slab feedback over the past few days. I will answer yours and Tejun's
> concerns at once:
>
> Here is the situation: the backtrace I get is this one:
>
> [124.956725] ================================
> [124.957217] [INFO: inconsistent lock state]
> [124.957217] 3.5.0+ #99 Not tainted
> [124.957217] --------------------------------
> [124.957217] inconsistent {SOFTIRQ-ON-W} -> {IN-SOFTIRQ-W} usage.
> [124.957217] ksoftirqd/0/3 [HC0[0]:SC1[1]:HE1:SE0] takes:
> [124.957217] (&(&ss->id_lock)->rlock){+.?...}, at:
> [<ffffffff810aa7b2>] spin_lock+0x9/0xb
> [124.957217] {SOFTIRQ-ON-W} state was registered at:
> [124.957217]   [<ffffffff810996ed>] __lock_acquire+0x31f/0xd68
> [124.957217]   [<ffffffff8109a660>] lock_acquire+0x108/0x15c
> [124.957217]   [<ffffffff81534ec4>] _raw_spin_lock+0x40/0x4f
> [124.957217]   [<ffffffff810aa7b2>] spin_lock+0x9/0xb
> [124.957217]   [<ffffffff810ad00e>] get_new_cssid+0x69/0xf3
> [124.957217]   [<ffffffff810ad0da>] cgroup_init_idr+0x42/0x60
> [124.957217]   [<ffffffff81b20e04>] cgroup_init+0x50/0x100
> [124.957217]   [<ffffffff81b05b9b>] start_kernel+0x3b9/0x3ee
> [124.957217]   [<ffffffff81b052d6>] x86_64_start_reservations+0xb1/0xb5
> [124.957217]   [<ffffffff81b053d8>] x86_64_start_kernel+0xfe/0x10b
>

>
> So what we learn from it, is: we are acquiring a specific lock (the css
> id one) from softirq context. It was previously taken in a
> softirq-enabled context, that seems to be coming directly from
> get_new_cssid.
>
> Tejun correctly pointed out that we should never acquire that lock from
> a softirq context, in which he is right.
>
> But the situation changes slightly with kmem. Now, the following excerpt
> of a backtrace is possible:
>
> [48.602775] [<ffffffff81103095>] free_accounted_pages+0x47/0x4c
> [48.602775] [<ffffffff81047f90>] free_task+0x31/0x5c
> [48.602775] [<ffffffff8104807d>] __put_task_struct+0xc2/0xdb
> [48.602775] [<ffffffff8104dfc7>] put_task_struct+0x1e/0x22
> [48.602775] [<ffffffff8104e144>] delayed_put_task_struct+0x7a/0x98
> [48.602775] [<ffffffff810cf0e5>] __rcu_process_callbacks+0x269/0x3df
> [48.602775] [<ffffffff810cf28c>] rcu_process_callbacks+0x31/0x5b
> [48.602775] [<ffffffff8105266d>] __do_softirq+0x122/0x277
>
> So as you can see, free_accounted_pages (that will trigger a memcg_put()
> -> mem_cgroup_free()) can now be called from softirq context, which is,
> an rcu callback (and I just realized I wrote the exact opposite in the
> subj line: man, I really suck at that!!)
> As a matter of fact, we could not move to our rcu callback as well:
>
> we need to move it to a worker thread with the rest.
>
> We already have a worker thread: he reason we have it is not
> static_branches: The reason is vfree(), that will BUG_ON(in_interrupt())
> and could not be called from rcu callback as well. We moved static
> branches in there as well for a similar problem, but haven't introduced it.
>
> Could we move just part of it to the worker thread? Absolutely yes.
> Moving just free_css_id() is enough to make it work. But since it is not
> the first context related problem we had, I thought: "to hell with that,
> let's move everything and be safe".
>
> I am fine moving free_css_id() only if you would prefer.
>
> Can we disable softirqs when we initialize css_id? Maybe. My machine
> seems to boot fine and survive the simple workload that would trigger
> that bug if I use irqsave spinlocks instead of normal spinlocks. But
> this has to be done from cgroup core: We have no control over css
> creation in memcg.
>
> How would you guys like me to handle this ?

Without the vfree callback, I would have preferred just making the id_lock softirq safe. But since we have to defer (parts of) freeing anyway, I like your approach of just deferring the rest as well better.

But please add comments why the stuff in there is actually deferred. Just simple notes like:

"this can be called from atomic contexts, <examples>",

"vfree must run from process context" and "css_id locking is not soft irq safe",

"to hell with that, let's just do everything from the workqueue and be safe and simple".

(And this may be personal preference, but why have free_work call __mem_cgroup_free()? Does anyone else need to call that code? There are too many layers already, why not just keep it all in free_work() and have one less stack frame on your mind? :))

As for the changelog, here is my attempt:

---

mm: memcg: defer whole memcg tear-down to workqueue

The final memcg put can already happen in atomic context and so the freeing is deferred to a workqueue because it needs to use vfree().

Kmem tracking will add freeing from softirq context, but the id_lock acquired when destroying the cgroup object is not softirq safe, e.g.:

> [48.602775] [<ffffffff81103095>] free_accounted_pages+0x47/0x4c
> [48.602775] [<ffffffff81047f90>] free_task+0x31/0x5c
> [48.602775] [<ffffffff8104807d>] __put_task_struct+0xc2/0xdb
> [48.602775] [<ffffffff8104dfc7>] put_task_struct+0x1e/0x22
> [48.602775] [<ffffffff8104e144>] delayed_put_task_struct+0x7a/0x98
> [48.602775] [<ffffffff810cf0e5>] __rcu_process_callbacks+0x269/0x3df
> [48.602775] [<ffffffff810cf28c>] rcu_process_callbacks+0x31/0x5b
> [48.602775] [<ffffffff8105266d>] __do_softirq+0x122/0x277

To avoid making tear-down too complicated - making locks soft irq safe, having half the cleanup in one function on the other half somewhere else - just defer everything to the workqueue.

Subject: Re: [PATCH v3 12/13] execute the whole memcg freeing in rcu callback
Posted by Glauber Costa on Mon, 08 Oct 2012 09:45:35 GMT
View Forum Message <> Reply to Message

On 10/05/2012 07:31 PM, Johannes Weiner wrote:
> On Thu, Oct 04, 2012 at 02:53:13PM +0400, Glauber Costa wrote:
>> On 10/01/2012 05:27 PM, Michal Hocko wrote:
>>> On Tue 18-09-12 18:04:09, Glauber Costa wrote:
>>>> A lot of the initialization we do in mem_cgroup_create() is done with softirqs
>>>> enabled. This include grabbing a css id, which holds &ss->id_lock->rlock, and
>>>> the per-zone trees, which holds rtpz->lock->rlock. All of those signal to the
>>>> lockdep mechanism that those locks can be used in SOFTIRQ-ON-W context. This
>>>> means that the freeing of memcg structure must happen in a compatible context,
>>>> otherwise we'll get a deadlock.
>>>
>>> Maybe I am missing something obvious but why cannot we simply disble
>>> (soft)irqs in mem_cgroup_create rather than make the free path much more
>>> complicated. It really feels strange to defer everything (e.g. soft
>>> reclaim tree cleanup which should be a no-op at the time because there
>>> shouldn't be any user pages in the group).
>>>
>>
>> Ok.
>>
>> I was just able to come back to this today - I was mostly working on the
>> slab feedback over the past few days. I will answer yours and Tejun's
>> concerns at once:
>>
>> Here is the situation: the backtrace I get is this one:
>>
>> [124.956725] ================================
>> [124.957217] [INFO: inconsistent lock state]
>> [124.957217] 3.5.0+ #99 Not tainted
>> [124.957217] --------------------------------
>> [124.957217] inconsistent {SOFTIRQ-ON-W} -> {IN-SOFTIRQ-W} usage.
>> [124.957217] ksoftirqd/0/3 [HC0[0]:SC1[1]:HE1:SE0] takes:
>> [124.957217] (&(&ss->id_lock)->rlock){+.?...}, at:
>> [<ffffffff810aa7b2>] spin_lock+0x9/0xb
>> [124.957217] {SOFTIRQ-ON-W} state was registered at:
>> [124.957217]   [<ffffffff810996ed>] __lock_acquire+0x31f/0xd68
>> [124.957217]   [<ffffffff8109a660>] lock_acquire+0x108/0x15c
>> [124.957217]   [<ffffffff81534ec4>] _raw_spin_lock+0x40/0x4f
>> [124.957217]   [<ffffffff810aa7b2>] spin_lock+0x9/0xb
>> [124.957217]   [<ffffffff810ad00e>] get_new_cssid+0x69/0xf3
>> [124.957217]   [<ffffffff810ad0da>] cgroup_init_idr+0x42/0x60
>> [124.957217]   [<ffffffff81b20e04>] cgroup_init+0x50/0x100
>> [124.957217]   [<ffffffff81b05b9b>] start_kernel+0x3b9/0x3ee
>> [124.957217]   [<ffffffff81b052d6>] x86_64_start_reservations+0xb1/0xb5
>> [124.957217]   [<ffffffff81b053d8>] x86_64_start_kernel+0xfe/0x10b

>>
>>
>> So what we learn from it, is: we are acquiring a specific lock (the css
>> id one) from softirq context. It was previously taken in a
>> softirq-enabled context, that seems to be coming directly from
>> get_new_cssid.
>>
>> Tejun correctly pointed out that we should never acquire that lock from
>> a softirq context, in which he is right.
>>
>> But the situation changes slightly with kmem. Now, the following excerpt
>> of a backtrace is possible:
>>
>> [48.602775] [<ffffffff81103095>] free_accounted_pages+0x47/0x4c
>> [48.602775] [<ffffffff81047f90>] free_task+0x31/0x5c
>> [48.602775] [<ffffffff8104807d>] __put_task_struct+0xc2/0xdb
>> [48.602775] [<ffffffff8104dfc7>] put_task_struct+0x1e/0x22
>> [48.602775] [<ffffffff8104e144>] delayed_put_task_struct+0x7a/0x98
>> [48.602775] [<ffffffff810cf0e5>] __rcu_process_callbacks+0x269/0x3df
>> [48.602775] [<ffffffff810cf28c>] rcu_process_callbacks+0x31/0x5b
>> [48.602775] [<ffffffff8105266d>] __do_softirq+0x122/0x277
>>
>> So as you can see, free_accounted_pages (that will trigger a memcg_put()
>> -> mem_cgroup_free()) can now be called from softirq context, which is,
>> an rcu callback (and I just realized I wrote the exact opposite in the
>> subj line: man, I really suck at that!!)
>> As a matter of fact, we could not move to our rcu callback as well:
>>
>> we need to move it to a worker thread with the rest.
>>
>> We already have a worker thread: he reason we have it is not
>> static_branches: The reason is vfree(), that will BUG_ON(in_interrupt())
>> and could not be called from rcu callback as well. We moved static
>> branches in there as well for a similar problem, but haven't introduced it.
>>
>> Could we move just part of it to the worker thread? Absolutely yes.
>> Moving just free_css_id() is enough to make it work. But since it is not
>> the first context related problem we had, I thought: "to hell with that,
>> let's move everything and be safe".
>>
>> I am fine moving free_css_id() only if you would prefer.
>>
>> Can we disable softirqs when we initialize css_id? Maybe. My machine
>> seems to boot fine and survive the simple workload that would trigger
>> that bug if I use irqsave spinlocks instead of normal spinlocks. But
>> this has to be done from cgroup core: We have no control over css
>> creation in memcg.
>>

>> How would you guys like me to handle this ?
>
> Without the vfree callback, I would have preferred just making the
> id_lock softirq safe. But since we have to defer (parts of) freeing
> anyway, I like your approach of just deferring the rest as well
> better.
>
> But please add comments why the stuff in there is actually deferred.
> Just simple notes like:
>
> "this can be called from atomic contexts, <examples>",
>
> "vfree must run from process context" and "css_id locking is not soft
> irq safe",
>
> "to hell with that, let's just do everything from the workqueue and be
> safe and simple".
>
> (And this may be personal preference, but why have free_work call
> __mem_cgroup_free()? Does anyone else need to call that code? There
> are too many layers already, why not just keep it all in free_work()
> and have one less stack frame on your mind? :))
>
It is used when create fails.
