Subject: [PATCH v4 00/25] kmem limitation for memcg
Posted by Glauber Costa on Mon, 18 Jun 2012 10:27:53 GMT
View Forum Message <> Reply to Message

Hello All,

This is my new take for the memcg kmem accounting. This should merge
all of the previous comments from you guys, specially concerning the big churn
inside the allocators themselves.

My focus in this new round was to keep the changes in the cache internals to
a minimum. To do that, I relied upon two main pillars:

* Cristoph's unification series, that allowed me to put must of the changes
  in a common file. Even then, the changes are not too many, since the overal
  level of invasiveness was decreased.
* Accounting is done directly from the page allocator. This means some pages
  can fail to be accounted, but that can only happen when the task calling
  kmem_cache_alloc or kmalloc is not the same task allocating a new page.
  This never happens in steady state operation if the tasks are kept in the
  same memcg. Naturally, if the page ends up being accounted to a memcg that
  is not limited (such as root memcg), that particular page will simply not
  be accounted.

The dispatcher code stays (mem_cgroup_get_kmem_cache), being the mechanism who
guarantees that, during steady state operation, all objects allocated in a page
will belong to the same memcg. I consider this a good compromise point between
strict and loose accounting here.

I should point out again that most, if not all, of the code in the caches
are wrapped in static_key areas, meaning they will be completely patched out
until the first limit is set. Enabling and disabling of static_keys incorporate
the last fixes for sock memcg, and should be pretty robust.

[v4]
* I decided to leave the file memory.kmem.slabinfo out of this series. It can
  be done a lot better if the caches have a common interface for that, which
  seems to be work in progress.
* Accounting is done directly from the page allocator.
* more code moved out of the cache internals.
[v3]
* fixed lockdep bugs in slab (ordering of get_online_cpus() vs slab_mutex)
* improved style in slab and slub with less #ifdefs in-code
* tested and fixed hierarchical accounting (memcg: propagate kmem limiting...)
* some more small bug fixes
* No longer using res_counter_charge_nofail for GFP_NOFAIL submissions. Those
  go to the root memcg directly.
* reordered tests in mem_cgroup_get_kmem_cache so we exit even earlier for

tasks in root memcg
* no more memcg state for slub initialization
* do_tune_cpucache will always (only after FULL) propagate to children when they exist.
* slab itself will destroy the kmem_cache string for chained caches, so we don't need to bother with consistency between them.
* other minor issues

[v2]
* memcgs can be properly removed.
* We are not charging based on current->mm->owner instead of current
* kmem_large allocations for slub got some fixes, specially for the free case
* A cache that is registered can be properly removed (common module case) even if it spans memcg children. Slab had some code for that, now it works well with both
* A new mechanism for skipping allocations is proposed (patch posted separately already). Now instead of having kmalloc_no_account, we mark a region as non-accountable for memcg.

Glauber Costa (23):
slab: rename gfpflags to allocflags
provide a common place for initcall processing in kmem_cache
slab: move FULL state transition to an initcall
Wipe out CFLGS_OFF_SLAB from flags during initial slab creation
memcg: Always free struct memcg through schedule_work()
memcg: change defines to an enum
kmem slab accounting basic infrastructure
slab/slub: struct memcg_params
consider a memcg parameter in kmem_create_cache
sl[au]b: always get the cache from its page in kfree
Add a __GFP_SLABMEMCG flag
memcg: kmem controller dispatch infrastructure
allow enable_cpu_cache to use preset values for its tunables
don't do __ClearPageSlab before freeing slab page.
skip memcg kmem allocations in specified code regions
mm: Allocate kernel pages to the right memcg
memcg: disable kmem code when not in use.
memcg: destroy memcg caches
Track all the memcg children of a kmem_cache.
slab: slab-specific propagation changes.
memcg: propagate kmem limiting information to children
memcg/slub: shrink dead caches
Documentation: add documentation for slab tracker for memcg

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

```
 Documentation/cgroups/memory.txt |   33 ++
 include/linux/gfp.h              |    8 +-
 include/linux/memcontrol.h       |  100 +++++
 include/linux/page-flags.h       |    2 +-
 include/linux/sched.h            |    1 +
 include/linux/slab.h             |   25 ++
 include/linux/slab_def.h         |    6 +-
 include/linux/slub_def.h         |   29 +-
 init/Kconfig                     |    2 +-
 mm/memcontrol.c                  |  874 ++++++++++++++++++++++++++++++++++++++--
 mm/page_alloc.c                  |   16 +-
 mm/slab.c                        |   94 +++-
 mm/slab.h                        |   57 ++-
 mm/slab_common.c                 |   72 +++-
 mm/slob.c                        |    6 +-
 mm/slub.c                        |   32 +-
 16 files changed, 1272 insertions(+), 85 deletions(-)

--
1.7.10.2
```

Subject: [PATCH v4 01/25] slab: rename gfpflags to allocflags
Posted by Glauber Costa on Mon, 18 Jun 2012 10:27:54 GMT
View Forum Message <> Reply to Message

A consistent name with slub saves us an acessor function.
In both caches, this field represents the same thing. We would
like to use it from the mem_cgroup code.

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

```
 include/linux/slab_def.h |    2 +-
 mm/slab.c                |   10 +++++-----
 2 files changed, 6 insertions(+), 6 deletions(-)

diff --git a/include/linux/slab_def.h b/include/linux/slab_def.h
index 1d93f27..0c634fa 100644
--- a/include/linux/slab_def.h
+++ b/include/linux/slab_def.h
@@ -39,7 +39,7 @@ struct kmem_cache {
  unsigned int gfporder;

  /* force GFP flags, e.g. GFP_DMA */
- gfp_t gfpflags;
+ gfp_t allocflags;
```

```
  size_t colour;   /* cache colouring range */
  unsigned int colour_off; /* colour offset */
diff --git a/mm/slab.c b/mm/slab.c
index 2476ad4..020605f 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -1746,7 +1746,7 @@ static void *kmem_getpages(struct kmem_cache *cachep, gfp_t flags,
int nodeid)
  flags |= __GFP_COMP;
 #endif

- flags |= cachep->gfpflags;
+ flags |= cachep->allocflags;
  if (cachep->flags & SLAB_RECLAIM_ACCOUNT)
   flags |= __GFP_RECLAIMABLE;

@@ -2338,9 +2338,9 @@ int __kmem_cache_create(struct kmem_cache *cachep)
  cachep->colour = left_over / cachep->colour_off;
  cachep->slab_size = slab_size;
  cachep->flags = flags;
- cachep->gfpflags = 0;
+ cachep->allocflags = 0;
  if (CONFIG_ZONE_DMA_FLAG && (flags & SLAB_CACHE_DMA))
-  cachep->gfpflags |= GFP_DMA;
+  cachep->allocflags |= GFP_DMA;
  cachep->size = size;
  cachep->reciprocal_buffer_size = reciprocal_value(size);

@@ -2653,9 +2653,9 @@ static void kmem_flagcheck(struct kmem_cache *cachep, gfp_t flags)
 {
  if (CONFIG_ZONE_DMA_FLAG) {
   if (flags & GFP_DMA)
-   BUG_ON(!(cachep->gfpflags & GFP_DMA));
+   BUG_ON(!(cachep->allocflags & GFP_DMA));
   else
-   BUG_ON(cachep->gfpflags & GFP_DMA);
+   BUG_ON(cachep->allocflags & GFP_DMA);
  }
 }

--
1.7.10.2
```

Subject: [PATCH v4 02/25] provide a common place for initcall processing in kmem_cache

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

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

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
CC: David Rientjes <rientjes@google.com>
---
 mm/slab.c | 5 ++---
 mm/slab.h | 1 +
 mm/slab_common.c | 5 +++++
 mm/slob.c | 5 +++++
 mm/slub.c | 4 +---
 5 files changed, 14 insertions(+), 6 deletions(-)

diff --git a/mm/slab.c b/mm/slab.c
index 020605f..e174e50 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -853,7 +853,7 @@ static void __cpuinit start_cpu_timer(int cpu)
  struct delayed_work *reap_work = &per_cpu(slab_reap_work, cpu);

  /*
- * When this gets called from do_initcalls via cpucache_init(),
+ * When this gets called from do_initcalls via __kmem_cache_initcall(),
   * init_workqueues() has already run, so keventd will be setup
   * at that time.
   */
@@ -1666,7 +1666,7 @@ void __init kmem_cache_init_late(void)
  */
 }

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

@@ -1677,7 +1677,6 @@ static int __init cpucache_init(void)
   start_cpu_timer(cpu);
  return 0;
 }
-__initcall(cpucache_init);

```
 static noinline void
 slab_out_of_memory(struct kmem_cache *cachep, gfp_t gfpflags, int nodeid)
diff --git a/mm/slab.h b/mm/slab.h
index b44a8cc..19e17c7 100644
--- a/mm/slab.h
+++ b/mm/slab.h
@@ -37,6 +37,7 @@ unsigned long calculate_alignment(unsigned long flags,

 /* Functions provided by the slab allocators */
 int __kmem_cache_create(struct kmem_cache *s);
+int __kmem_cache_initcall(void);

 #ifdef CONFIG_SLUB
 struct kmem_cache *__kmem_cache_alias(const char *name, size_t size,
diff --git a/mm/slab_common.c b/mm/slab_common.c
index b5def07..15db694ac 100644
--- a/mm/slab_common.c
+++ b/mm/slab_common.c
@@ -203,3 +203,8 @@ int slab_is_available(void)
  return slab_state >= UP;
 }

+static int __init kmem_cache_initcall(void)
+{
+ return __kmem_cache_initcall();
+}
+__initcall(kmem_cache_initcall);
diff --git a/mm/slob.c b/mm/slob.c
index 507589d..61b1845 100644
--- a/mm/slob.c
+++ b/mm/slob.c
@@ -610,3 +610,8 @@ void __init kmem_cache_init(void)
 void __init kmem_cache_init_late(void)
 {
 }
+
+int __init kmem_cache_initcall(void)
+{
+ return 0;
+}
diff --git a/mm/slub.c b/mm/slub.c
index 7fc3499..d9d4d5a 100644
--- a/mm/slub.c
+++ b/mm/slub.c
@@ -5309,7 +5309,7 @@ static int sysfs_slab_alias(struct kmem_cache *s, const char *name)
  return 0;
 }
```

```
-static int __init slab_sysfs_init(void)
+int __init __kmem_cache_initcall(void)
 {
  struct kmem_cache *s;
  int err;
@@ -5347,8 +5347,6 @@ static int __init slab_sysfs_init(void)
  resiliency_test();
  return 0;
 }
-
-__initcall(slab_sysfs_init);
 #endif /* CONFIG_SYSFS */

 /*
--
1.7.10.2
```

Subject: [PATCH v4 03/25] slab: move FULL state transition to an initcall
Posted by Glauber Costa on Mon, 18 Jun 2012 10:27:56 GMT
View Forum Message <> Reply to Message

During kmem_cache_init_late(), we transition to the LATE state,
and after some more work, to the FULL state, its last state

This is quite different from slub, that will only transition to
its last state (previously SYSFS), in a (late)initcall, after a lot
more of the kernel is ready.

This means that in slab, we have no way to taking actions dependent
on the initialization of other pieces of the kernel that are supposed
to start way after kmem_init_late(), such as cgroups initialization.

To achieve more consistency in this behavior, that patch only
transitions to the UP state in kmem_init_late. In my analysis,
setup_cpu_cache() should be happy to test for >= UP, instead of
== FULL. It also has passed some tests I've made.

We then only mark FULL state after the reap timers are in place,
meaning that no further setup is expected.

Signed-off-by: Glauber Costa <glommer@parallels.com>
Acked-by: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
CC: David Rientjes <rientjes@google.com>
---
 mm/slab.c | 8 ++++----

```
 1 file changed, 4 insertions(+), 4 deletions(-)

diff --git a/mm/slab.c b/mm/slab.c
index e174e50..2d5fe28 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -1643,9 +1643,6 @@ void __init kmem_cache_init_late(void)
     BUG();
  mutex_unlock(&slab_mutex);

- /* Done! */
- slab_state = FULL;
-
  /*
   * Register a cpu startup notifier callback that initializes
   * cpu_cache_get for all new cpus
@@ -1675,6 +1672,9 @@ int __init __kmem_cache_initcall(void)
   */
  for_each_online_cpu(cpu)
   start_cpu_timer(cpu);
+
+ /* Done! */
+ slab_state = FULL;
  return 0;
 }

@@ -2120,7 +2120,7 @@ static size_t calculate_slab_order(struct kmem_cache *cachep,

 static int __init_refok setup_cpu_cache(struct kmem_cache *cachep, gfp_t gfp)
 {
- if (slab_state == FULL)
+ if (slab_state >= UP)
   return enable_cpucache(cachep, gfp);

  if (slab_state == DOWN) {
--
1.7.10.2
```

Subject: [PATCH v4 04/25] Wipe out CFLGS_OFF_SLAB from flags during initial slab creation

Posted by Glauber Costa on Mon, 18 Jun 2012 10:27:57 GMT

View Forum Message <> Reply to Message

CFLGS_OFF_SLAB is not a valid flag to be passed to cache creation.
If we are duplicating a cache - support added in a future patch -
we will rely on the flags it has stored in itself. That may include
CFLGS_OFF_SLAB.

So it is better to clean this flag at cache creation.

CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
CC: David Rientjes <rientjes@google.com>
---
 mm/slab.c | 6 ++++++
 1 file changed, 6 insertions(+)

diff --git a/mm/slab.c b/mm/slab.c
index 2d5fe28..c30a61c 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -2201,6 +2201,12 @@ int __kmem_cache_create(struct kmem_cache *cachep)
   BUG_ON(flags & SLAB_POISON);
 #endif
  /*
+ * Passing this flag at creation time is invalid, but if we're
+ * duplicating a slab, it may happen.
+ */
+ flags &= ~CFLGS_OFF_SLAB;
+
+ /*
   * Always checks flags, a caller might be expecting debug support which
   * isn't available.
   */
--
1.7.10.2

Subject: [PATCH v4 05/25] memcg: Always free struct memcg through schedule_work()
Posted by Glauber Costa on Mon, 18 Jun 2012 10:27:58 GMT
View Forum Message <> Reply to Message

Right now we free struct memcg with kfree right after a
rcu grace period, but defer it if we need to use vfree() to get
rid of that memory area. We do that by need, because we need vfree
to be called in a process context.

This patch unifies this behavior, by ensuring that even kfree will
happen in a separate thread. The goal is to have a stable place to
call the upcoming jump label destruction function outside the realm
of the complicated and quite far-reaching cgroup lock (that can't be
held when calling neither the cpu_hotplug.lock nor the jump_label_mutex)

Signed-off-by: Glauber Costa <glommer@parallels.com>

```
CC: Tejun Heo <tj@kernel.org>
CC: Li Zefan <lizefan@huawei.com>
CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Johannes Weiner <hannes@cmpxchg.org>
CC: Michal Hocko <mhocko@suse.cz>
---
 mm/memcontrol.c |   24 ++++++++++++++++++-----------
 1 file changed, 13 insertions(+), 11 deletions(-)

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index e3b528e..ce15be4 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -245,8 +245,8 @@ struct mem_cgroup {
    */
   struct rcu_head rcu_freeing;
   /*
-   * But when using vfree(), that cannot be done at
-   * interrupt time, so we must then queue the work.
+   * We also need some space for a worker in deferred freeing.
+   * By the time we call it, rcu_freeing is not longer in use.
    */
   struct work_struct work_freeing;
 };
@@ -4826,23 +4826,28 @@ out_free:
 }

 /*
- * Helpers for freeing a vzalloc()ed mem_cgroup by RCU,
+ * 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 vfree_work(struct work_struct *work)
+static void free_work(struct work_struct *work)
 {
  struct mem_cgroup *memcg;
+ int size = sizeof(struct mem_cgroup);

  memcg = container_of(work, struct mem_cgroup, work_freeing);
- vfree(memcg);
+ if (size < PAGE_SIZE)
+  kfree(memcg);
+ else
+  vfree(memcg);
 }
-static void vfree_rcu(struct rcu_head *rcu_head)
+
```

```
+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, vfree_work);
+ INIT_WORK(&memcg->work_freeing, free_work);
  schedule_work(&memcg->work_freeing);
 }

@@ -4868,10 +4873,7 @@ static void __mem_cgroup_free(struct mem_cgroup *memcg)
   free_mem_cgroup_per_zone_info(memcg, node);

  free_percpu(memcg->stat);
- if (sizeof(struct mem_cgroup) < PAGE_SIZE)
-  kfree_rcu(memcg, rcu_freeing);
- else
-  call_rcu(&memcg->rcu_freeing, vfree_rcu);
+ call_rcu(&memcg->rcu_freeing, free_rcu);
 }

 static void mem_cgroup_get(struct mem_cgroup *memcg)
--
1.7.10.2
```

Subject: [PATCH v4 06/25] memcg: Make it possible to use the stock for more than one page.
Posted by Glauber Costa on Mon, 18 Jun 2012 10:27:59 GMT
View Forum Message <> Reply to Message

From: Suleiman Souhlal <ssouhlal@FreeBSD.org>

Signed-off-by: Suleiman Souhlal <suleiman@google.com>
Signed-off-by: Glauber Costa <glommer@parallels.com>
Acked-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
---
 mm/memcontrol.c | 18 +++++++++---------
 1 file changed, 9 insertions(+), 9 deletions(-)

```
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index ce15be4..00b9f1e 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -1998,19 +1998,19 @@ 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.
+ * Try to consume stocked charge on this cpu. If success, nr_pages pages are
+ * 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.
  */
-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;

  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);
@@ -2309,7 +2309,7 @@ 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 {
@@ -2334,7 +2334,7 @@ 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
--
1.7.10.2
```

Subject: [PATCH v4 07/25] memcg: Reclaim when more than one page needed.

From: Suleiman Souhlal <ssouhlal@FreeBSD.org>

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

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

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

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

Signed-off-by: Suleiman Souhlal <suleiman@google.com>
Signed-off-by: Glauber Costa <glommer@parallels.com>
Reviewed-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
---
 mm/memcontrol.c | 18 ++++++++++-------
 1 file changed, 11 insertions(+), 7 deletions(-)

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 00b9f1e..b6cb075 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -2187,7 +2187,8 @@ enum {
 };

 static int mem_cgroup_do_charge(struct mem_cgroup *memcg, gfp_t gfp_mask,
-    unsigned int nr_pages, bool oom_check)
+   unsigned int nr_pages, unsigned int min_pages,
+   bool oom_check)
 {
  unsigned long csize = nr_pages * PAGE_SIZE;
  struct mem_cgroup *mem_over_limit;
@@ -2210,18 +2211,18 @@ static int mem_cgroup_do_charge(struct mem_cgroup *memcg,
gfp_t gfp_mask,
  } else
   mem_over_limit = mem_cgroup_from_res_counter(fail_res, res);
  /*
- * nr_pages can be either a huge page (HPAGE_PMD_NR), a batch

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

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

+ if (gfp_mask & __GFP_NORETRY)
+  return CHARGE_NOMEM;
+
  ret = mem_cgroup_reclaim(mem_over_limit, gfp_mask, flags);
  if (mem_cgroup_margin(mem_over_limit) >= nr_pages)
   return CHARGE_RETRY;
@@ -2234,8 +2235,10 @@ static int mem_cgroup_do_charge(struct mem_cgroup *memcg, gfp_t
gfp_mask,
   * unlikely to succeed so close to the limit, and we fall back
   * to regular pages anyway in case of failure.
   */
- if (nr_pages == 1 && ret)
+ if (nr_pages <= (1 << PAGE_ALLOC_COSTLY_ORDER) && ret) {
+  cond_resched();
   return CHARGE_RETRY;
+ }

  /*
   * At task move, charge accounts can be doubly counted. So, it's
@@ -2369,7 +2372,8 @@ again:
    nr_oom_retries = MEM_CGROUP_RECLAIM_RETRIES;
   }

-  ret = mem_cgroup_do_charge(memcg, gfp_mask, batch, oom_check);
+  ret = mem_cgroup_do_charge(memcg, gfp_mask, batch, nr_pages,
+      oom_check);
   switch (ret) {
   case CHARGE_OK:
    break;
--
1.7.10.2
```

Subject: [PATCH v4 08/25] memcg: change defines to an enum
Posted by Glauber Costa on Mon, 18 Jun 2012 10:28:01 GMT

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.

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Michal Hocko <mhocko@suse.cz>
CC: Johannes Weiner <hannes@cmpxchg.org>
Acked-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
---
 mm/memcontrol.c | 9 ++++++---
 1 file changed, 6 insertions(+), 3 deletions(-)

```
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index b6cb075..cc1fdb4 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -374,9 +374,12 @@ 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)
--
1.7.10.2
```

Subject: [PATCH v4 09/25] kmem slab accounting basic infrastructure
Posted by Glauber Costa on Mon, 18 Jun 2012 10:28:02 GMT

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

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

 * memory.kmem.max_usage_in_bytes

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

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

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

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

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Michal Hocko <mhocko@suse.cz>
CC: Johannes Weiner <hannes@cmpxchg.org>
Reviewed-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
---
 mm/memcontrol.c | 78 ++-
 1 file changed, 77 insertions(+), 1 deletion(-)

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index cc1fdb4..04ac774 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -252,6 +252,10 @@ struct mem_cgroup {
 };

 /*
+ * the counter to account for kernel memory usage.
+ */
+struct res_counter kmem;
+/*
  * Per cgroup active and inactive list, similar to the
  * per zone LRU lists.
  */
@@ -266,6 +270,7 @@ struct mem_cgroup {
  * Should the accounting and control be hierarchical, per subtree?
  */
 bool use_hierarchy;
+bool kmem_accounted;

 bool oom_lock;

```
  atomic_t under_oom;
@@ -378,6 +383,7 @@ enum res_type {
  _MEM,
  _MEMSWAP,
  _OOM_TYPE,
+ _KMEM,
 };

 #define MEMFILE_PRIVATE(x, val) (((x) << 16) | (val))
@@ -1470,6 +1476,10 @@ done:
   res_counter_read_u64(&memcg->memsw, RES_USAGE) >> 10,
   res_counter_read_u64(&memcg->memsw, RES_LIMIT) >> 10,
   res_counter_read_u64(&memcg->memsw, RES_FAILCNT));
+ printk(KERN_INFO "kmem: usage %llukB, limit %llukB, failcnt %llu\n",
+  res_counter_read_u64(&memcg->kmem, RES_USAGE) >> 10,
+  res_counter_read_u64(&memcg->kmem, RES_LIMIT) >> 10,
+  res_counter_read_u64(&memcg->kmem, RES_FAILCNT));
 }

 /*
@@ -3908,6 +3918,11 @@ static ssize_t mem_cgroup_read(struct cgroup *cont, struct cftype
*cft,
   else
    val = res_counter_read_u64(&memcg->memsw, name);
   break;
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ case _KMEM:
+  val = res_counter_read_u64(&memcg->kmem, name);
+  break;
+#endif
  default:
   BUG();
  }
@@ -3945,8 +3960,26 @@ static int mem_cgroup_write(struct cgroup *cont, struct cftype *cft,
    break;
   if (type == _MEM)
    ret = mem_cgroup_resize_limit(memcg, val);
-  else
+  else if (type == _MEMSWAP)
    ret = mem_cgroup_resize_memsw_limit(memcg, val);
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+  else if (type == _KMEM) {
+   ret = res_counter_set_limit(&memcg->kmem, val);
+   if (ret)
+    break;
+   /*
+    * Once enabled, can't be disabled. We could in theory
+    * disable it if we haven't yet created any caches, or
```

```
+    * if we can shrink them all to death.
+    *
+    * But it is not worth the trouble
+    */
+   if (!memcg->kmem_accounted && val != RESOURCE_MAX)
+    memcg->kmem_accounted = true;
+  }
+#endif
+  else
+   return -EINVAL;
    break;
   case RES_SOFT_LIMIT:
    ret = res_counter_memparse_write_strategy(buffer, &val);
@@ -4011,12 +4044,20 @@ static int mem_cgroup_reset(struct cgroup *cont, unsigned int
event)
   case RES_MAX_USAGE:
    if (type == _MEM)
    res_counter_reset_max(&memcg->res);
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+  else if (type == _KMEM)
+   res_counter_reset_max(&memcg->kmem);
+#endif
    else
    res_counter_reset_max(&memcg->memsw);
    break;
   case RES_FAILCNT:
    if (type == _MEM)
    res_counter_reset_failcnt(&memcg->res);
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+  else if (type == _KMEM)
+   res_counter_reset_failcnt(&memcg->kmem);
+#endif
    else
    res_counter_reset_failcnt(&memcg->memsw);
    break;
@@ -4647,6 +4688,33 @@ static int mem_control_numa_stat_open(struct inode *unused, struct
file *file)
 #endif /* CONFIG_NUMA */

 #ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+static struct cftype kmem_cgroup_files[] = {
+ {
+  .name = "kmem.limit_in_bytes",
+  .private = MEMFILE_PRIVATE(_KMEM, RES_LIMIT),
+  .write_string = mem_cgroup_write,
+  .read = mem_cgroup_read,
+ },
+ {
```

```
+  .name = "kmem.usage_in_bytes",
+  .private = MEMFILE_PRIVATE(_KMEM, RES_USAGE),
+  .read = mem_cgroup_read,
+ },
+ {
+  .name = "kmem.failcnt",
+  .private = MEMFILE_PRIVATE(_KMEM, RES_FAILCNT),
+  .trigger = mem_cgroup_reset,
+  .read = mem_cgroup_read,
+ },
+ {
+  .name = "kmem.max_usage_in_bytes",
+  .private = MEMFILE_PRIVATE(_KMEM, RES_MAX_USAGE),
+  .trigger = mem_cgroup_reset,
+  .read = mem_cgroup_read,
+ },
+ {},
+};
+
 static int memcg_init_kmem(struct mem_cgroup *memcg, struct cgroup_subsys *ss)
 {
  return mem_cgroup_sockets_init(memcg, ss);
@@ -4981,6 +5049,12 @@ mem_cgroup_create(struct cgroup *cont)
   int cpu;
   enable_swap_cgroup();
   parent = NULL;
+
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+  WARN_ON(cgroup_add_cftypes(&mem_cgroup_subsys,
+        kmem_cgroup_files));
+#endif
+
   if (mem_cgroup_soft_limit_tree_init())
    goto free_out;
   root_mem_cgroup = memcg;
@@ -4999,6 +5073,7 @@ mem_cgroup_create(struct cgroup *cont)
  if (parent && parent->use_hierarchy) {
   res_counter_init(&memcg->res, &parent->res);
   res_counter_init(&memcg->memsw, &parent->memsw);
+  res_counter_init(&memcg->kmem, &parent->kmem);
   /*
    * We increment refcnt of the parent to ensure that we can
    * safely access it on res_counter_charge/uncharge.
@@ -5009,6 +5084,7 @@ mem_cgroup_create(struct cgroup *cont)
  } else {
   res_counter_init(&memcg->res, NULL);
   res_counter_init(&memcg->memsw, NULL);
+  res_counter_init(&memcg->kmem, NULL);
```

```
 }
 memcg->last_scanned_node = MAX_NUMNODES;
 INIT_LIST_HEAD(&memcg->oom_notify);
--
1.7.10.2
```

Subject: [PATCH v4 10/25] slab/slub: struct memcg_params
Posted by Glauber Costa on Mon, 18 Jun 2012 10:28:03 GMT
View Forum Message <> Reply to Message

For the kmem slab controller, we need to record some extra
information in the kmem_cache structure.

Signed-off-by: Glauber Costa <glommer@parallels.com>
Signed-off-by: Suleiman Souhlal <suleiman@google.com>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
CC: Michal Hocko <mhocko@suse.cz>
CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Johannes Weiner <hannes@cmpxchg.org>
---
 include/linux/slab.h | 8 ++++++++
 include/linux/slab_def.h | 4 ++++
 include/linux/slub_def.h | 3 +++
 3 files changed, 15 insertions(+)

diff --git a/include/linux/slab.h b/include/linux/slab.h
index 3c2181a..e007cc5 100644
--- a/include/linux/slab.h
+++ b/include/linux/slab.h
@@ -177,6 +177,14 @@ unsigned int kmem_cache_size(struct kmem_cache *);
 #define ARCH_SLAB_MINALIGN __alignof__(unsigned long long)
 #endif

+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+struct mem_cgroup_cache_params {
+ struct mem_cgroup *memcg;
+ int id;
+ atomic_t refcnt;
+};
+#endif
+
 /*
  * Common kmalloc functions provided by all allocators
  */
diff --git a/include/linux/slab_def.h b/include/linux/slab_def.h
index 0c634fa..04ca5be 100644

```
--- a/include/linux/slab_def.h
+++ b/include/linux/slab_def.h
@@ -83,6 +83,10 @@ struct kmem_cache {
  int obj_offset;
 #endif /* CONFIG_DEBUG_SLAB */

+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ struct mem_cgroup_cache_params memcg_params;
+#endif
+
 /* 6) per-cpu/per-node data, touched during every alloc/free */
  /*
   * We put array[] at the end of kmem_cache, because we want to size
diff --git a/include/linux/slub_def.h b/include/linux/slub_def.h
index df448ad..7637f3b 100644
--- a/include/linux/slub_def.h
+++ b/include/linux/slub_def.h
@@ -101,6 +101,9 @@ struct kmem_cache {
 #ifdef CONFIG_SYSFS
  struct kobject kobj; /* For sysfs */
 #endif
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ struct mem_cgroup_cache_params memcg_params;
+#endif

 #ifdef CONFIG_NUMA
  /*
--
1.7.10.2
```

Subject: [PATCH v4 11/25] consider a memcg parameter in kmem_create_cache
Posted by Glauber Costa on Mon, 18 Jun 2012 10:28:04 GMT
View Forum Message <> Reply to Message

Allow a memcg parameter to be passed during cache creation.
When the slub allocator is being used, it will only merge
caches that belong to the same memcg.

Default function is created as a wrapper, passing NULL
to the memcg version. We only merge caches that belong
to the same memcg.

>From the memcontrol.c side, 3 helper functions are created:

1) memcg_css_id: because slub needs a unique cache name
   for sysfs. Since this is visible, but not the canonical
   location for slab data, the cache name is not used, the

css_id should suffice.

2) mem_cgroup_register_cache: is responsible for assigning
a unique index to each cache, and other general purpose
setup. The index is only assigned for the root caches. All
others are assigned index == -1.

3) mem_cgroup_release_cache: can be called from the root cache
destruction, and will release the index for
other caches.

We can't assign indexes until the basic slab is up and running
this is because the ida subsystem will itself call slab functions
such as kmalloc a couple of times. Because of that, we have
a late_initcall that scan all caches and register them after the
kernel is booted up. Only caches registered after that receive
their index right away.

This index mechanism was developed by Suleiman Souhlal.
Changed to a idr/ida based approach based on suggestion
from Kamezawa.

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
CC: Michal Hocko <mhocko@suse.cz>
CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Johannes Weiner <hannes@cmpxchg.org>
CC: Suleiman Souhlal <suleiman@google.com>
---
include/linux/memcontrol.h | 14 +++++++++++++
include/linux/slab.h | 10 ++++++++++
mm/memcontrol.c | 25 ++++++++++++++++++++++++
mm/slab.h | 22 ++++++++++++++++++++++-----
mm/slab_common.c | 36 ++++++++++++++++++++++++++++++++++++-----
mm/slub.c | 16 ++++++++++++++++----
6 files changed, 109 insertions(+), 14 deletions(-)

diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
index f94efd2..99e14b9 100644
--- a/include/linux/memcontrol.h
+++ b/include/linux/memcontrol.h
@@ -26,6 +26,7 @@ struct mem_cgroup;
struct page_cgroup;
struct page;
struct mm_struct;
+struct kmem_cache;

```
 /* Stats that can be updated by kernel. */
 enum mem_cgroup_page_stat_item {
@@ -440,7 +441,20 @@ struct sock;
 #ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
 void sock_update_memcg(struct sock *sk);
 void sock_release_memcg(struct sock *sk);
+int memcg_css_id(struct mem_cgroup *memcg);
+void mem_cgroup_register_cache(struct mem_cgroup *memcg,
+          struct kmem_cache *s);
+void mem_cgroup_release_cache(struct kmem_cache *cachep);
 #else
+static inline void mem_cgroup_register_cache(struct mem_cgroup *memcg,
+          struct kmem_cache *s)
+{
+}
+
+static inline void mem_cgroup_release_cache(struct kmem_cache *cachep)
+{
+}
+
 static inline void sock_update_memcg(struct sock *sk)
 {
 }
diff --git a/include/linux/slab.h b/include/linux/slab.h
index e007cc5..d347616 100644
--- a/include/linux/slab.h
+++ b/include/linux/slab.h
@@ -116,6 +116,7 @@ struct kmem_cache {
 };
 #endif

+struct mem_cgroup;
 /*
  * struct kmem_cache related prototypes
  */
@@ -125,6 +126,9 @@ int slab_is_available(void);
 struct kmem_cache *kmem_cache_create(const char *, size_t, size_t,
    unsigned long,
    void (*)(void *));
+struct kmem_cache *
+kmem_cache_create_memcg(struct mem_cgroup *, const char *, size_t, size_t,
+   unsigned long, void (*)(void *));
 void kmem_cache_destroy(struct kmem_cache *);
 int kmem_cache_shrink(struct kmem_cache *);
 void kmem_cache_free(struct kmem_cache *, void *);
@@ -338,6 +342,12 @@ extern void *__kmalloc_track_caller(size_t, gfp_t, unsigned long);
  __kmalloc(size, flags)
 #endif /* DEBUG_SLAB */
```

```
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+#define MAX_KMEM_CACHE_TYPES 400
+#else
+#define MAX_KMEM_CACHE_TYPES 0
+#endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
+
 #ifdef CONFIG_NUMA
 /*
  * kmalloc_node_track_caller is a special version of kmalloc_node that
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 04ac774..cb57c5b 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -323,6 +323,11 @@ struct mem_cgroup {
 #endif
 };

+int memcg_css_id(struct mem_cgroup *memcg)
+{
+ return css_id(&memcg->css);
+}
+
 /* Stuffs for move charges at task migration. */
 /*
  * Types of charges to be moved. "move_charge_at_immitgrate" is treated as a
@@ -461,6 +466,25 @@ struct cg_proto *tcp_proto_cgroup(struct mem_cgroup *memcg)
 }
 EXPORT_SYMBOL(tcp_proto_cgroup);
 #endif /* CONFIG_INET */
+
+struct ida cache_types;
+
+void mem_cgroup_register_cache(struct mem_cgroup *memcg,
+          struct kmem_cache *cachep)
+{
+ int id = -1;
+
+ if (!memcg)
+  id = ida_simple_get(&cache_types, 0, MAX_KMEM_CACHE_TYPES,
+        GFP_KERNEL);
+ cachep->memcg_params.id = id;
+}
+
+void mem_cgroup_release_cache(struct kmem_cache *cachep)
+{
+ if (cachep->memcg_params.id != -1)
+  ida_simple_remove(&cache_types, cachep->memcg_params.id);
```

```
+}
 #endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */

 static void drain_all_stock_async(struct mem_cgroup *memcg);
@@ -5053,6 +5077,7 @@ mem_cgroup_create(struct cgroup *cont)
 #ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
   WARN_ON(cgroup_add_cftypes(&mem_cgroup_subsys,
         kmem_cgroup_files));
+  ida_init(&cache_types);
 #endif

   if (mem_cgroup_soft_limit_tree_init())
diff --git a/mm/slab.h b/mm/slab.h
index 19e17c7..1781580 100644
--- a/mm/slab.h
+++ b/mm/slab.h
@@ -39,16 +39,28 @@ unsigned long calculate_alignment(unsigned long flags,
 int __kmem_cache_create(struct kmem_cache *s);
 int __kmem_cache_initcall(void);

+struct mem_cgroup;
 #ifdef CONFIG_SLUB
-struct kmem_cache *__kmem_cache_alias(const char *name, size_t size,
- size_t align, unsigned long flags, void (*ctor)(void *));
+struct kmem_cache *
+__kmem_cache_alias(struct mem_cgroup *memcg, const char *name, size_t size,
+     size_t align, unsigned long flags, void (*ctor)(void *));
 #else
-static inline struct kmem_cache *__kmem_cache_alias(const char *name, size_t size,
- size_t align, unsigned long flags, void (*ctor)(void *))
+static inline struct kmem_cache *
+__kmem_cache_alias(struct mem_cgroup *memcg, const char *name, size_t size,
+     size_t align, unsigned long flags, void (*ctor)(void *))
 { return NULL; }
 #endif

-
 int __kmem_cache_shutdown(struct kmem_cache *);

+static inline bool cache_match_memcg(struct kmem_cache *cachep,
+         struct mem_cgroup *memcg)
+{
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+        return cachep->memcg_params.memcg == memcg;
+#endif
+        return true;
+}
+
```

```
+void __init memcg_slab_register_all(void);
 #endif
diff --git a/mm/slab_common.c b/mm/slab_common.c
index 15db694ac..42f226d 100644
--- a/mm/slab_common.c
+++ b/mm/slab_common.c
@@ -16,6 +16,7 @@
 #include <asm/cacheflush.h>
 #include <asm/tlbflush.h>
 #include <asm/page.h>
+#include <linux/memcontrol.h>

 #include "slab.h"

@@ -77,8 +78,9 @@ unsigned long calculate_alignment(unsigned long flags,
  * as davem.
  */

-struct kmem_cache *kmem_cache_create(const char *name, size_t size, size_t align,
-  unsigned long flags, void (*ctor)(void *))
+struct kmem_cache *
+kmem_cache_create_memcg(struct mem_cgroup *memcg, const char *name, size_t size,
+   size_t align, unsigned long flags, void (*ctor)(void *))
 {
  struct kmem_cache *s = NULL;
  char *n;
@@ -118,7 +120,7 @@ struct kmem_cache *kmem_cache_create(const char *name, size_t size,
size_t align
    continue;
   }

-  if (!strcmp(s->name, name)) {
+  if (cache_match_memcg(s, memcg) && !strcmp(s->name, name)) {
    printk(KERN_ERR "kmem_cache_create(%s): Cache name"
     " already exists.\n",
     name);
@@ -131,7 +133,7 @@ struct kmem_cache *kmem_cache_create(const char *name, size_t size,
size_t align
  WARN_ON(strchr(name, ' ')); /* It confuses parsers */
 #endif

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

@@ -152,11 +154,17 @@ struct kmem_cache *kmem_cache_create(const char *name, size_t
size, size_t align
```

```
  s->flags = flags;
  s->align = calculate_alignment(flags, align, size);

+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ s->memcg_params.memcg = memcg;
+#endif
+
  r = __kmem_cache_create(s);

  if (!r) {
   s->refcount = 1;
   list_add(&s->list, &slab_caches);
+  if (slab_state >= FULL)
+   mem_cgroup_register_cache(memcg, s);
  }
  else {
   kmem_cache_free(kmem_cache, s);
@@ -173,6 +181,12 @@ oops:

  return s;
 }
+
+struct kmem_cache *kmem_cache_create(const char *name, size_t size, size_t align,
+  unsigned long flags, void (*ctor)(void *))
+{
+ return kmem_cache_create_memcg(NULL, name, size, align, flags, ctor);
+}
 EXPORT_SYMBOL(kmem_cache_create);

 void kmem_cache_destroy(struct kmem_cache *s)
@@ -185,6 +199,7 @@ void kmem_cache_destroy(struct kmem_cache *s)
   if (s->flags & SLAB_DESTROY_BY_RCU)
    rcu_barrier();

+  mem_cgroup_release_cache(s);
   kfree(s->name);
   kmem_cache_free(kmem_cache, s);
  } else {
@@ -205,6 +220,17 @@ int slab_is_available(void)

 static int __init kmem_cache_initcall(void)
 {
- return __kmem_cache_initcall();
+ int r = __kmem_cache_initcall();
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ struct kmem_cache *s;
+
+ if (r)
```

```
+  return r;
+ mutex_lock(&slab_mutex);
+ list_for_each_entry(s, &slab_caches, list)
+  mem_cgroup_register_cache(NULL, s);
+ mutex_unlock(&slab_mutex);
+#endif
+ return r;
 }
 __initcall(kmem_cache_initcall);
diff --git a/mm/slub.c b/mm/slub.c
index d9d4d5a..ca4b8e0 100644
--- a/mm/slub.c
+++ b/mm/slub.c
@@ -31,6 +31,7 @@
 #include <linux/fault-inject.h>
 #include <linux/stacktrace.h>
 #include <linux/prefetch.h>
+#include <linux/memcontrol.h>

 #include <trace/events/kmem.h>

@@ -3831,7 +3832,7 @@ static int slab_unmergeable(struct kmem_cache *s)
  return 0;
 }

-static struct kmem_cache *find_mergeable(size_t size,
+static struct kmem_cache *find_mergeable(struct mem_cgroup *memcg, size_t size,
   size_t align, unsigned long flags, const char *name,
   void (*ctor)(void *))
 {
@@ -3867,17 +3868,20 @@ static struct kmem_cache *find_mergeable(size_t size,
   if (s->size - size >= sizeof(void *))
    continue;

+  if (!cache_match_memcg(s, memcg))
+   continue;
   return s;
  }
  return NULL;
 }

-struct kmem_cache *__kmem_cache_alias(const char *name, size_t size,
-  size_t align, unsigned long flags, void (*ctor)(void *))
+struct kmem_cache *
+__kmem_cache_alias(struct mem_cgroup *memcg, const char *name, size_t size,
+     size_t align, unsigned long flags, void (*ctor)(void *))
 {
  struct kmem_cache *s;
```

```
- s = find_mergeable(size, align, flags, name, ctor);
+ s = find_mergeable(memcg, size, align, flags, name, ctor);
  if (s) {
   s->refcount++;
   /*
@@ -5207,6 +5211,10 @@ static char *create_unique_id(struct kmem_cache *s)
  if (p != name + 1)
   *p++ = '-';
  p += sprintf(p, "%07d", s->size);
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ if (s->memcg_params.memcg)
+  p += sprintf(p, "-%08d", memcg_css_id(s->memcg_params.memcg));
+#endif
  BUG_ON(p > name + ID_STR_LENGTH - 1);
  return name;
 }
--
1.7.10.2
```

Subject: [PATCH v4 12/25] sl[au]b: always get the cache from its page in kfree
Posted by Glauber Costa on Mon, 18 Jun 2012 10:28:05 GMT
View Forum Message <> Reply to Message

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

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

[v3: added parent testing with VM_BUG_ON]

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
---
 mm/slab.c | 5 ++++-
 mm/slab.h | 10 ++++++++++
 mm/slub.c | 3 ++-
 3 files changed, 16 insertions(+), 2 deletions(-)

diff --git a/mm/slab.c b/mm/slab.c
index c30a61c..3783a6a 100644
--- a/mm/slab.c
+++ b/mm/slab.c

```
@@ -3712,9 +3712,12 @@ EXPORT_SYMBOL(__kmalloc);
  * Free an object which was previously allocated from this
  * cache.
  */
-void kmem_cache_free(struct kmem_cache *cachep, void *objp)
+void kmem_cache_free(struct kmem_cache *s, void *objp)
 {
  unsigned long flags;
+ struct kmem_cache *cachep = virt_to_cache(objp);
+
+ VM_BUG_ON(!((s == cachep) | slab_is_parent(s, cachep)));

  local_irq_save(flags);
  debug_check_no_locks_freed(objp, cachep->size);
diff --git a/mm/slab.h b/mm/slab.h
index 1781580..0a3e712 100644
--- a/mm/slab.h
+++ b/mm/slab.h
@@ -63,4 +63,14 @@ static inline bool cache_match_memcg(struct kmem_cache *cachep,
 }

 void __init memcg_slab_register_all(void);
+
+static inline bool slab_is_parent(struct kmem_cache *s,
+      struct kmem_cache *candidate)
+{
+#if defined(CONFIG_CGROUP_MEM_RES_CTLR_KMEM) && defined(CONFIG_DEBUG_VM)
+ return candidate == s->memcg_params.parent;
+#else
+ return false;
+#endif
+}
 #endif
diff --git a/mm/slub.c b/mm/slub.c
index ca4b8e0..e685cfa 100644
--- a/mm/slub.c
+++ b/mm/slub.c
@@ -2597,7 +2597,8 @@ void kmem_cache_free(struct kmem_cache *s, void *x)

  page = virt_to_head_page(x);

- slab_free(s, page, x, _RET_IP_);
+ VM_BUG_ON(!((page->slab == s) | slab_is_parent(page->slab, s)));
+ slab_free(page->slab, page, x, _RET_IP_);

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

Subject: [PATCH v4 13/25] Add a __GFP_SLABMEMCG flag
Posted by Glauber Costa on Mon, 18 Jun 2012 10:28:06 GMT
View Forum Message <> Reply to Message

This flag is used to indicate to the callees that this allocation will be
serviced to the kernel. It is not supposed to be passed by the callers
of kmem_cache_alloc, but rather by the cache core itself.

CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
CC: Michal Hocko <mhocko@suse.cz>
CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Johannes Weiner <hannes@cmpxchg.org>
CC: Suleiman Souhlal <suleiman@google.com>
---
 include/linux/gfp.h | 8 +++++++-
 1 file changed, 7 insertions(+), 1 deletion(-)

diff --git a/include/linux/gfp.h b/include/linux/gfp.h
index 581e74b..f904fe1 100644
--- a/include/linux/gfp.h
+++ b/include/linux/gfp.h
@@ -37,6 +37,9 @@ struct vm_area_struct;
 #define ___GFP_NO_KSWAPD 0x400000u
 #define ___GFP_OTHER_NODE 0x800000u
 #define ___GFP_WRITE 0x1000000u
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+#define ___GFP_SLABMEMCG 0x2000000u
+#endif

 /*
  * GFP bitmasks..
@@ -88,13 +91,16 @@ 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 */

+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+#define __GFP_SLABMEMCG ((__force gfp_t)___GFP_SLABMEMCG)/* Allocation comes from a memcg slab */
+#endif
 /*
  * This may seem redundant, but it's a way of annotating false positives vs.
  * allocations that simply cannot be supported (e.g. page tables).
  */

```
 #define __GFP_NOTRACK_FALSE_POSITIVE (__GFP_NOTRACK)

-#define __GFP_BITS_SHIFT 25 /* Room for N __GFP_FOO bits */
+#define __GFP_BITS_SHIFT 26 /* Room for N __GFP_FOO bits */
 #define __GFP_BITS_MASK ((__force gfp_t)((1 << __GFP_BITS_SHIFT) - 1))

 /* This equals 0, but use constants in case they ever change */
--
1.7.10.2
```

Subject: [PATCH v4 14/25] memcg: kmem controller dispatch infrastructure
Posted by Glauber Costa on Mon, 18 Jun 2012 10:28:07 GMT
View Forum Message <> Reply to Message

With all the dependencies already in place, this patch introduces
the dispatcher functions for the slab cache accounting in memcg.

Before we can charge a cache, we need to select the right cache.
This is done by using the function __mem_cgroup_get_kmem_cache().

If we should use the root kmem cache, this function tries to detect
that and return as early as possible.

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

Because the slub allocator tends to inline the allocations whenever
it can, those functions need to be exported so modules can make use
of it properly.

This code is inspired by the code written by Suleiman Souhlal,
but heavily changed.

Signed-off-by: Glauber Costa <glommer@parallels.com>
Signed-off-by: Suleiman Souhlal <suleiman@google.com>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
CC: Michal Hocko <mhocko@suse.cz>
CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Johannes Weiner <hannes@cmpxchg.org>

```
 include/linux/memcontrol.h |   80 +++++++++
 init/Kconfig               |    2 +-
 mm/memcontrol.c            |  390 ++++++++++++++++++++++++++++++++++++++++++++-
 3 files changed, 470 insertions(+), 2 deletions(-)
```

```
diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
index 99e14b9..27a3f16 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;
@@ -445,6 +446,19 @@ int memcg_css_id(struct mem_cgroup *memcg);
 void mem_cgroup_register_cache(struct mem_cgroup *memcg,
                struct kmem_cache *s);
 void mem_cgroup_release_cache(struct kmem_cache *cachep);
+void mem_cgroup_flush_cache_create_queue(void);
+bool __mem_cgroup_new_kmem_page(gfp_t gfp, void *handle, int order);
+void __mem_cgroup_commit_kmem_page(struct page *page, void *handle, int order);
+void __mem_cgroup_free_kmem_page(struct page *page, int order);
+
+#define mem_cgroup_kmem_on 1
+struct kmem_cache *
+__mem_cgroup_get_kmem_cache(struct kmem_cache *cachep, gfp_t gfp);
+
+static inline bool has_memcg_flag(gfp_t gfp)
+{
+ return gfp & __GFP_SLABMEMCG;
+}
 #else
 static inline void mem_cgroup_register_cache(struct mem_cgroup *memcg,
                struct kmem_cache *s)
@@ -461,6 +475,72 @@ static inline void sock_update_memcg(struct sock *sk)
 static inline void sock_release_memcg(struct sock *sk)
 {
 }
+
+static inline void
+mem_cgroup_flush_cache_create_queue(void)
+{
+}
+
+static inline void mem_cgroup_destroy_cache(struct kmem_cache *cachep)
+{
+}
+
+static inline bool has_memcg_flag(gfp_t gfp)
+{
```

```
+ return false;
+}
+
+#define mem_cgroup_kmem_on 0
+#define __mem_cgroup_get_kmem_cache(a, b) a
+#define __mem_cgroup_new_kmem_page(a, b, c) false
+#define __mem_cgroup_free_kmem_page(a,b )
+#define __mem_cgroup_commit_kmem_page(a, b, c)
 #endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
+static __always_inline struct kmem_cache *
+mem_cgroup_get_kmem_cache(struct kmem_cache *cachep, gfp_t gfp)
+{
+ if (!mem_cgroup_kmem_on)
+  return cachep;
+ if (!current->mm)
+  return cachep;
+ if (in_interrupt())
+  return cachep;
+ if (gfp & __GFP_NOFAIL)
+  return cachep;
+
+ return __mem_cgroup_get_kmem_cache(cachep, gfp);
+}
+
+static __always_inline
+bool mem_cgroup_new_kmem_page(gfp_t gfp, void *handle, int order)
+{
+ if (!mem_cgroup_kmem_on)
+  return true;
+ if (!has_memcg_flag(gfp))
+  return true;
+ if (!current->mm)
+  return true;
+ if (in_interrupt())
+  return true;
+ if (gfp & __GFP_NOFAIL)
+  return true;
+ return __mem_cgroup_new_kmem_page(gfp, handle, order);
+}
+
+static __always_inline
+void mem_cgroup_free_kmem_page(struct page *page, int order)
+{
+ if (mem_cgroup_kmem_on)
+  __mem_cgroup_free_kmem_page(page, order);
+}
+
+static __always_inline
```

```
+void mem_cgroup_commit_kmem_page(struct page *page, struct mem_cgroup *handle,
+     int order)
+{
+ if (mem_cgroup_kmem_on)
+  __mem_cgroup_commit_kmem_page(page, handle, order);
+}
+
 #endif /* _LINUX_MEMCONTROL_H */

diff --git a/init/Kconfig b/init/Kconfig
index 6cfd71d..af98c30 100644
--- a/init/Kconfig
+++ b/init/Kconfig
@@ -696,7 +696,7 @@ config CGROUP_MEM_RES_CTLR_SWAP_ENABLED
    then swapaccount=0 does the trick).
 config CGROUP_MEM_RES_CTLR_KMEM
  bool "Memory Resource Controller Kernel Memory accounting (EXPERIMENTAL)"
- depends on CGROUP_MEM_RES_CTLR && EXPERIMENTAL
+ depends on CGROUP_MEM_RES_CTLR && EXPERIMENTAL && !SLOB
  default n
  help
    The Kernel Memory extension for Memory Resource Controller can limit
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index cb57c5b..beead5e 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
@@ -321,6 +325,11 @@ struct mem_cgroup {
 #ifdef CONFIG_INET
  struct tcp_memcontrol tcp_mem;
 #endif
+
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ /* Slab accounting */
+ struct kmem_cache *slabs[MAX_KMEM_CACHE_TYPES];
+#endif
 };
```

```
 int memcg_css_id(struct mem_cgroup *memcg)
@@ -414,6 +423,9 @@ static void mem_cgroup_put(struct mem_cgroup *memcg);
 #include <net/ip.h>

 static bool mem_cgroup_is_root(struct mem_cgroup *memcg);
+static int memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t gfp, s64 delta);
+static void memcg_uncharge_kmem(struct mem_cgroup *memcg, s64 delta);
+
 void sock_update_memcg(struct sock *sk)
 {
  if (mem_cgroup_sockets_enabled) {
@@ -467,7 +479,48 @@ struct cg_proto *tcp_proto_cgroup(struct mem_cgroup *memcg)
 EXPORT_SYMBOL(tcp_proto_cgroup);
 #endif /* CONFIG_INET */

+static char *mem_cgroup_cache_name(struct mem_cgroup *memcg, struct kmem_cache
*cachep)
+{
+ char *name;
+ struct dentry *dentry;
+
+ rcu_read_lock();
+ dentry = rcu_dereference(memcg->css.cgroup->dentry);
+ rcu_read_unlock();
+
+ BUG_ON(dentry == NULL);
+
+ name = kasprintf(GFP_KERNEL, "%s(%d:%s)",
+     cachep->name, css_id(&memcg->css), dentry->d_name.name);
+
+ return name;
+}
+
+static struct kmem_cache *kmem_cache_dup(struct mem_cgroup *memcg,
+      struct kmem_cache *s)
+{
+ char *name;
+ struct kmem_cache *new;
+
+ name = mem_cgroup_cache_name(memcg, s);
+ if (!name)
+  return NULL;
+
+ new = kmem_cache_create_memcg(memcg, name, s->object_size, s->align,
+          (s->flags & ~SLAB_PANIC), s->ctor);
+
+ kfree(name);
+ return new;
```

```
+}
+
+static inline bool mem_cgroup_kmem_enabled(struct mem_cgroup *memcg)
+{
+ return !mem_cgroup_disabled() && memcg &&
+        !mem_cgroup_is_root(memcg) && memcg->kmem_accounted;
+}
+
 struct ida cache_types;
+static DEFINE_MUTEX(memcg_cache_mutex);

 void mem_cgroup_register_cache(struct mem_cgroup *memcg,
           struct kmem_cache *cachep)
@@ -482,9 +535,274 @@ void mem_cgroup_register_cache(struct mem_cgroup *memcg,

 void mem_cgroup_release_cache(struct kmem_cache *cachep)
 {
+ mem_cgroup_flush_cache_create_queue();
   if (cachep->memcg_params.id != -1)
    ida_simple_remove(&cache_types, cachep->memcg_params.id);
 }
+
+static struct kmem_cache *memcg_create_kmem_cache(struct mem_cgroup *memcg,
+        struct kmem_cache *cachep)
+{
+ struct kmem_cache *new_cachep;
+ int idx;
+
+ BUG_ON(!mem_cgroup_kmem_enabled(memcg));
+
+ idx = cachep->memcg_params.id;
+
+ mutex_lock(&memcg_cache_mutex);
+ new_cachep = memcg->slabs[idx];
+ if (new_cachep)
+  goto out;
+
+ new_cachep = kmem_cache_dup(memcg, cachep);
+
+ if (new_cachep == NULL) {
+  new_cachep = cachep;
+  goto out;
+ }
+
+ mem_cgroup_get(memcg);
+ memcg->slabs[idx] = new_cachep;
+ new_cachep->memcg_params.memcg = memcg;
+ atomic_set(&new_cachep->memcg_params.refcnt, 1);
```

```
+out:
+ mutex_unlock(&memcg_cache_mutex);
+ return new_cachep;
+}
+
+struct create_work {
+ struct mem_cgroup *memcg;
+ struct kmem_cache *cachep;
+ struct list_head list;
+};
+
+/* Use a single spinlock for destruction and creation, not a frequent op */
+static DEFINE_SPINLOCK(cache_queue_lock);
+static LIST_HEAD(create_queue);
+
+/*
+ * Flush the queue of kmem_caches to create, because we're creating a cgroup.
+ *
+ * We might end up flushing other cgroups' creation requests as well, but
+ * they will just get queued again next time someone tries to make a slab
+ * allocation for them.
+ */
+void mem_cgroup_flush_cache_create_queue(void)
+{
+ struct create_work *cw, *tmp;
+ unsigned long flags;
+
+ spin_lock_irqsave(&cache_queue_lock, flags);
+ list_for_each_entry_safe(cw, tmp, &create_queue, list) {
+  list_del(&cw->list);
+  kfree(cw);
+ }
+ spin_unlock_irqrestore(&cache_queue_lock, flags);
+}
+
+static void memcg_create_cache_work_func(struct work_struct *w)
+{
+ struct create_work *cw, *tmp;
+ unsigned long flags;
+ LIST_HEAD(create_unlocked);
+
+ spin_lock_irqsave(&cache_queue_lock, flags);
+ list_for_each_entry_safe(cw, tmp, &create_queue, list)
+  list_move(&cw->list, &create_unlocked);
+ spin_unlock_irqrestore(&cache_queue_lock, flags);
+
+ list_for_each_entry_safe(cw, tmp, &create_unlocked, list) {
+  list_del(&cw->list);
```

```
+  memcg_create_kmem_cache(cw->memcg, cw->cachep);
+  /* Drop the reference gotten when we enqueued. */
+  css_put(&cw->memcg->css);
+  kfree(cw);
+ }
+}
+
+static DECLARE_WORK(memcg_create_cache_work, memcg_create_cache_work_func);
+
+/*
+ * Enqueue the creation of a per-memcg kmem_cache.
+ * Called with rcu_read_lock.
+ */
+static void memcg_create_cache_enqueue(struct mem_cgroup *memcg,
+           struct kmem_cache *cachep)
+{
+ struct create_work *cw;
+ unsigned long flags;
+
+ spin_lock_irqsave(&cache_queue_lock, flags);
+ list_for_each_entry(cw, &create_queue, list) {
+  if (cw->memcg == memcg && cw->cachep == cachep) {
+   spin_unlock_irqrestore(&cache_queue_lock, flags);
+   return;
+  }
+ }
+ spin_unlock_irqrestore(&cache_queue_lock, flags);
+
+ /* The corresponding put will be done in the workqueue. */
+ if (!css_tryget(&memcg->css))
+  return;
+
+ cw = kmalloc(sizeof(struct create_work), GFP_NOWAIT);
+ if (cw == NULL) {
+  css_put(&memcg->css);
+  return;
+ }
+
+ cw->memcg = memcg;
+ cw->cachep = cachep;
+ spin_lock_irqsave(&cache_queue_lock, flags);
+ list_add_tail(&cw->list, &create_queue);
+ spin_unlock_irqrestore(&cache_queue_lock, flags);
+
+ schedule_work(&memcg_create_cache_work);
+}
+
+/*
```

```
+ * Return the kmem_cache we're supposed to use for a slab allocation.
+ * We try to use the current memcg's version of the cache.
+ *
+ * If the cache does not exist yet, if we are the first user of it,
+ * we either create it immediately, if possible, or create it asynchronously
+ * in a workqueue.
+ * In the latter case, we will let the current allocation go through with
+ * the original cache.
+ *
+ * Can't be called in interrupt context or from kernel threads.
+ * This function needs to be called with rcu_read_lock() held.
+ */
+struct kmem_cache *__mem_cgroup_get_kmem_cache(struct kmem_cache *cachep,
+          gfp_t gfp)
+{
+ struct mem_cgroup *memcg;
+ int idx;
+ struct task_struct *p;
+
+ if (cachep->memcg_params.memcg)
+  return cachep;
+
+ idx = cachep->memcg_params.id;
+ VM_BUG_ON(idx == -1);
+
+ rcu_read_lock();
+ p = rcu_dereference(current->mm->owner);
+ memcg = mem_cgroup_from_task(p);
+ rcu_read_unlock();
+
+ if (!mem_cgroup_kmem_enabled(memcg))
+  return cachep;
+
+ if (memcg->slabs[idx] == NULL) {
+  memcg_create_cache_enqueue(memcg, cachep);
+  return cachep;
+ }
+
+ return memcg->slabs[idx];
+}
+EXPORT_SYMBOL(__mem_cgroup_get_kmem_cache);
+
+bool __mem_cgroup_new_kmem_page(gfp_t gfp, void *_handle, int order)
+{
+ struct mem_cgroup *memcg;
+ struct mem_cgroup *handle = *(struct mem_cgroup **)_handle;
+ bool ret = true;
+ size_t size;
```

```
+ struct task_struct *p;
+
+ handle = NULL;
+ rcu_read_lock();
+ p = rcu_dereference(current->mm->owner);
+ memcg = mem_cgroup_from_task(p);
+ if (!mem_cgroup_kmem_enabled(memcg))
+  goto out;
+
+ mem_cgroup_get(memcg);
+
+ size = (1 << order) << PAGE_SHIFT;
+ ret = memcg_charge_kmem(memcg, gfp, size) == 0;
+ if (!ret) {
+  mem_cgroup_put(memcg);
+  goto out;
+ }
+ handle = memcg;
+out:
+ rcu_read_unlock();
+ return ret;
+}
+EXPORT_SYMBOL(__mem_cgroup_new_kmem_page);
+
+void __mem_cgroup_commit_kmem_page(struct page *page, void *handle, int order)
+{
+ struct page_cgroup *pc;
+ struct mem_cgroup *memcg = handle;
+ size_t size;
+
+ if (!memcg)
+  return;
+
+ WARN_ON(mem_cgroup_is_root(memcg));
+ /* The page allocation must have failed. Revert */
+ if (!page) {
+  size = (1 << order) << PAGE_SHIFT;
+  memcg_uncharge_kmem(memcg, size);
+  mem_cgroup_put(memcg);
+  return;
+ }
+
+ pc = lookup_page_cgroup(page);
+ lock_page_cgroup(pc);
+ pc->mem_cgroup = memcg;
+ SetPageCgroupUsed(pc);
+ unlock_page_cgroup(pc);
+}
```

```
+void __mem_cgroup_free_kmem_page(struct page *page, int order)
+{
+ struct mem_cgroup *memcg;
+ size_t size;
+ struct page_cgroup *pc;
+
+ if (mem_cgroup_disabled())
+  return;
+
+ pc = lookup_page_cgroup(page);
+ lock_page_cgroup(pc);
+ memcg = pc->mem_cgroup;
+ pc->mem_cgroup = NULL;
+ if (!PageCgroupUsed(pc)) {
+  unlock_page_cgroup(pc);
+  return;
+ }
+ ClearPageCgroupUsed(pc);
+ unlock_page_cgroup(pc);
+
+ /*
+  * The classical disabled check won't work
+  * for uncharge, since it is possible that the user enabled
+  * kmem tracking, allocated, and then disabled.
+  *
+  * We trust if there is a memcg associated with the page,
+  * it is a valid allocation
+  */
+ if (!memcg)
+  return;
+
+ WARN_ON(mem_cgroup_is_root(memcg));
+ size = (1 << order) << PAGE_SHIFT;
+ memcg_uncharge_kmem(memcg, size);
+ mem_cgroup_put(memcg);
+}
+EXPORT_SYMBOL(__mem_cgroup_free_kmem_page);
+
+static void memcg_slab_init(struct mem_cgroup *memcg)
+{
+ int i;
+
+ for (i = 0; i < MAX_KMEM_CACHE_TYPES; i++)
+  memcg->slabs[i] = NULL;
+}
 #endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */

 static void drain_all_stock_async(struct mem_cgroup *memcg);
```

```
@@ -4741,7 +5059,11 @@ static struct cftype kmem_cgroup_files[] = {

 static int memcg_init_kmem(struct mem_cgroup *memcg, struct cgroup_subsys *ss)
 {
- return mem_cgroup_sockets_init(memcg, ss);
+ int ret = mem_cgroup_sockets_init(memcg, ss);
+
+ if (!ret)
+  memcg_slab_init(memcg);
+ return ret;
 };

 static void kmem_cgroup_destroy(struct mem_cgroup *memcg)
@@ -5757,3 +6079,69 @@ static int __init enable_swap_account(char *s)
 __setup("swapaccount=", enable_swap_account);

 #endif
+
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+int memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t gfp, s64 delta)
+{
+ struct res_counter *fail_res;
+ struct mem_cgroup *_memcg;
+ int may_oom, ret;
+ bool nofail = false;
+
+ may_oom = (gfp & __GFP_WAIT) && (gfp & __GFP_FS) &&
+     !(gfp & __GFP_NORETRY);
+
+ ret = 0;
+
+ if (!memcg)
+  return ret;
+
+ _memcg = memcg;
+ ret = __mem_cgroup_try_charge(NULL, gfp, delta / PAGE_SIZE,
+     &_memcg, may_oom);
+
+ if (ret == -EINTR)  {
+  nofail = true;
+  /*
+   * __mem_cgroup_try_charge() chose to bypass to root due
+   * to OOM kill or fatal signal.
+   * Since our only options are to either fail the
+   * allocation or charge it to this cgroup, do it as
+   * a temporary condition. But we can't fail. From a kmem/slab
+   * perspective, the cache has already been selected, by
+   * mem_cgroup_get_kmem_cache(), so it is too late to change our
```

```
+   * minds
+   */
+  res_counter_charge_nofail(&memcg->res, delta, &fail_res);
+  if (do_swap_account)
+   res_counter_charge_nofail(&memcg->memsw, delta,
+         &fail_res);
+  ret = 0;
+ } else if (ret == -ENOMEM)
+  return ret;
+
+ if (nofail)
+  res_counter_charge_nofail(&memcg->kmem, delta, &fail_res);
+ else
+  ret = res_counter_charge(&memcg->kmem, delta, &fail_res);
+
+ if (ret) {
+  res_counter_uncharge(&memcg->res, delta);
+  if (do_swap_account)
+   res_counter_uncharge(&memcg->memsw, delta);
+ }
+
+ return ret;
+}
+
+void memcg_uncharge_kmem(struct mem_cgroup *memcg, s64 delta)
+{
+ if (!memcg)
+  return;
+
+ res_counter_uncharge(&memcg->kmem, delta);
+ res_counter_uncharge(&memcg->res, delta);
+ if (do_swap_account)
+  res_counter_uncharge(&memcg->memsw, delta);
+}
+#endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
--
1.7.10.2
```

Subject: [PATCH v4 15/25] allow enable_cpu_cache to use preset values for its tunables

Posted by Glauber Costa on Mon, 18 Jun 2012 10:28:08 GMT

View Forum Message <> Reply to Message

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

This could be done by an explicit call to do_tune_cpucache() after the
cache is created. But this is not very convenient now that the caches are
created from common code, since this function is SLAB-specific.

Another method of doing that is taking advantage of the fact that
do_tune_cpucache() is always called from enable_cpucache(), which is
called at cache initialization. We can just preset the values, and
then things work as expected.

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
CC: Michal Hocko <mhocko@suse.cz>
CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Johannes Weiner <hannes@cmpxchg.org>
CC: Suleiman Souhlal <suleiman@google.com>
---
 include/linux/slab.h | 3 ++-
 mm/memcontrol.c | 2 +-
 mm/slab.c | 19 ++++++++++++++++---
 mm/slab_common.c | 7 ++++---
 4 files changed, 23 insertions(+), 8 deletions(-)

diff --git a/include/linux/slab.h b/include/linux/slab.h
index d347616..d2d2fad 100644
--- a/include/linux/slab.h
+++ b/include/linux/slab.h
@@ -128,7 +128,7 @@ struct kmem_cache *kmem_cache_create(const char *, size_t, size_t,
    void (*)(void *));
 struct kmem_cache *
 kmem_cache_create_memcg(struct mem_cgroup *, const char *, size_t, size_t,
-   unsigned long, void (*)(void *));
+   unsigned long, void (*)(void *), struct kmem_cache *);
 void kmem_cache_destroy(struct kmem_cache *);
 int kmem_cache_shrink(struct kmem_cache *);
 void kmem_cache_free(struct kmem_cache *, void *);
@@ -184,6 +184,7 @@ unsigned int kmem_cache_size(struct kmem_cache *);
 #ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
 struct mem_cgroup_cache_params {
  struct mem_cgroup *memcg;
+ struct kmem_cache *parent;
  int id;
  atomic_t refcnt;
 };
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index beead5e..324e550 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c

```
@@ -507,7 +507,7 @@ static struct kmem_cache *kmem_cache_dup(struct mem_cgroup
*memcg,
   return NULL;

  new = kmem_cache_create_memcg(memcg, name, s->object_size, s->align,
-         (s->flags & ~SLAB_PANIC), s->ctor);
+         (s->flags & ~SLAB_PANIC), s->ctor, s);

  kfree(name);
  return new;
diff --git a/mm/slab.c b/mm/slab.c
index 3783a6a..c548666 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -3918,8 +3918,19 @@ static int do_tune_cpucache(struct kmem_cache *cachep, int limit,
 static int enable_cpucache(struct kmem_cache *cachep, gfp_t gfp)
 {
  int err;
- int limit, shared;
-
+ int limit = 0;
+ int shared = 0;
+ int batchcount = 0;
+
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+       if (cachep->memcg_params.parent) {
+               limit = cachep->memcg_params.parent->limit;
+               shared = cachep->memcg_params.parent->shared;
+               batchcount = cachep->memcg_params.parent->batchcount;
+       }
+#endif
+ if (limit && shared && batchcount)
+  goto skip_setup;
  /*
   * The head array serves three purposes:
   * - create a LIFO ordering, i.e. return objects that are cache-warm
@@ -3961,7 +3972,9 @@ static int enable_cpucache(struct kmem_cache *cachep, gfp_t gfp)
  if (limit > 32)
   limit = 32;
 #endif
- err = do_tune_cpucache(cachep, limit, (limit + 1) / 2, shared, gfp);
+ batchcount = (limit + 1) / 2;
+skip_setup:
+ err = do_tune_cpucache(cachep, limit, batchcount, shared, gfp);
  if (err)
   printk(KERN_ERR "enable_cpucache failed for %s, error %d.\n",
         cachep->name, -err);
diff --git a/mm/slab_common.c b/mm/slab_common.c
```

```
index 42f226d..619d365 100644
--- a/mm/slab_common.c
+++ b/mm/slab_common.c
@@ -80,7 +80,8 @@ unsigned long calculate_alignment(unsigned long flags,

 struct kmem_cache *
 kmem_cache_create_memcg(struct mem_cgroup *memcg, const char *name, size_t size,
-   size_t align, unsigned long flags, void (*ctor)(void *))
+   size_t align, unsigned long flags, void (*ctor)(void *),
+   struct kmem_cache *parent_cache)
 {
  struct kmem_cache *s = NULL;
  char *n;
@@ -153,9 +154,9 @@ kmem_cache_create_memcg(struct mem_cgroup *memcg, const char
*name, size_t size,
  s->ctor = ctor;
  s->flags = flags;
  s->align = calculate_alignment(flags, align, size);
-
 #ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
  s->memcg_params.memcg = memcg;
+ s->memcg_params.parent = parent_cache;
 #endif

  r = __kmem_cache_create(s);
@@ -185,7 +186,7 @@ oops:
 struct kmem_cache *kmem_cache_create(const char *name, size_t size, size_t align,
   unsigned long flags, void (*ctor)(void *))
 {
- return kmem_cache_create_memcg(NULL, name, size, align, flags, ctor);
+ return kmem_cache_create_memcg(NULL, name, size, align, flags, ctor, NULL);
 }
 EXPORT_SYMBOL(kmem_cache_create);

--
1.7.10.2
```

Subject: [PATCH v4 16/25] don't do __ClearPageSlab before freeing slab page.
Posted by Glauber Costa on Mon, 18 Jun 2012 10:28:09 GMT

View Forum Message <> Reply to Message

This will give the oportunity to the page allocator to
determine that a given page was previously a slab page, and
take action accordingly.

If memcg kmem is present, this means that that page needs to
be unaccounted. The page allocator will now have the responsibility

to clear that bit upon free_pages().

It is not uncommon to have the page allocator to check page flags.
Mlock flag, for instance, is checked pervasively all over the place.
So I hope this is okay for the slab as well.

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
CC: Michal Hocko <mhocko@suse.cz>
CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Johannes Weiner <hannes@cmpxchg.org>
CC: Suleiman Souhlal <suleiman@google.com>
---
 mm/page_alloc.c | 5 ++++-
 mm/slab.c | 5 -----
 mm/slob.c | 1 -
 mm/slub.c | 1 -
 4 files changed, 4 insertions(+), 8 deletions(-)

diff --git a/mm/page_alloc.c b/mm/page_alloc.c
index 918330f..a884a9c 100644
--- a/mm/page_alloc.c
+++ b/mm/page_alloc.c
@@ -697,8 +697,10 @@ static bool free_pages_prepare(struct page *page, unsigned int order)

  if (PageAnon(page))
   page->mapping = NULL;
- for (i = 0; i < (1 << order); i++)
+ for (i = 0; i < (1 << order); i++) {
+ __ClearPageSlab(page + i);
   bad += free_pages_check(page + i);
+ }
  if (bad)
   return false;

@@ -2505,6 +2507,7 @@ EXPORT_SYMBOL(get_zeroed_page);
 void __free_pages(struct page *page, unsigned int order)
 {
  if (put_page_testzero(page)) {
+ __ClearPageSlab(page);
   if (order == 0)
    free_hot_cold_page(page, 0);
   else
diff --git a/mm/slab.c b/mm/slab.c
index c548666..e537406 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -1795,11 +1795,6 @@ static void kmem_freepages(struct kmem_cache *cachep, void

```
*addr)
  else
   sub_zone_page_state(page_zone(page),
     NR_SLAB_UNRECLAIMABLE, nr_freed);
- while (i--) {
-  BUG_ON(!PageSlab(page));
-  __ClearPageSlab(page);
-  page++;
- }
  if (current->reclaim_state)
   current->reclaim_state->reclaimed_slab += nr_freed;
  free_pages((unsigned long)addr, cachep->gfporder);
diff --git a/mm/slob.c b/mm/slob.c
index 61b1845..b03d65e 100644
--- a/mm/slob.c
+++ b/mm/slob.c
@@ -360,7 +360,6 @@ static void slob_free(void *block, int size)
   if (slob_page_free(sp))
    clear_slob_page_free(sp);
   spin_unlock_irqrestore(&slob_lock, flags);
-  __ClearPageSlab(sp);
   reset_page_mapcount(sp);
   slob_free_pages(b, 0);
   return;
diff --git a/mm/slub.c b/mm/slub.c
index e685cfa..69c5677 100644
--- a/mm/slub.c
+++ b/mm/slub.c
@@ -1399,7 +1399,6 @@ static void __free_slab(struct kmem_cache *s, struct page *page)
   NR_SLAB_RECLAIMABLE : NR_SLAB_UNRECLAIMABLE,
   -pages);

- __ClearPageSlab(page);
  reset_page_mapcount(page);
  if (current->reclaim_state)
   current->reclaim_state->reclaimed_slab += pages;
--
1.7.10.2
```

Subject: [PATCH v4 17/25] skip memcg kmem allocations in specified code regions
Posted by Glauber Costa on Mon, 18 Jun 2012 10:28:10 GMT
View Forum Message <> Reply to Message

This patch creates a mechanism that skip memcg allocations during
certain pieces of our core code. It basically works in the same way
as preempt_disable()/preempt_enable(): By marking a region under
which all allocations will be accounted to the root memcg.

We need this to prevent races in early cache creation, when we
allocate data using caches that are not necessarily created already.

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
CC: Michal Hocko <mhocko@suse.cz>
CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Johannes Weiner <hannes@cmpxchg.org>
CC: Suleiman Souhlal <suleiman@google.com>
---
 include/linux/sched.h | 1 +
 mm/memcontrol.c | 26 ++++++++++++++++++++++++++
 2 files changed, 27 insertions(+)

diff --git a/include/linux/sched.h b/include/linux/sched.h
index 81a173c..0761dda 100644
--- a/include/linux/sched.h
+++ b/include/linux/sched.h
@@ -1613,6 +1613,7 @@ struct task_struct {
   unsigned long nr_pages; /* uncharged usage */
   unsigned long memsw_nr_pages; /* uncharged mem+swap usage */
  } memcg_batch;
+ unsigned int memcg_kmem_skip_account;
 #endif
 #ifdef CONFIG_HAVE_HW_BREAKPOINT
  atomic_t ptrace_bp_refcnt;
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 324e550..233d3bc 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -479,6 +479,22 @@ struct cg_proto *tcp_proto_cgroup(struct mem_cgroup *memcg)
 EXPORT_SYMBOL(tcp_proto_cgroup);
 #endif /* CONFIG_INET */

+static void memcg_stop_kmem_account(void)
+{
+ if (!current->mm)
+ return;
+
+ current->memcg_kmem_skip_account++;
+}
+
+static void memcg_resume_kmem_account(void)
+{
+ if (!current->mm)
+ return;

```
+
+ current->memcg_kmem_skip_account--;
+}
+
 static char *mem_cgroup_cache_name(struct mem_cgroup *memcg, struct kmem_cache
*cachep)
 {
  char *name;
@@ -555,7 +571,9 @@ static struct kmem_cache *memcg_create_kmem_cache(struct
mem_cgroup *memcg,
  if (new_cachep)
   goto out;

+ memcg_stop_kmem_account();
  new_cachep = kmem_cache_dup(memcg, cachep);
+ memcg_resume_kmem_account();

  if (new_cachep == NULL) {
   new_cachep = cachep;
@@ -646,7 +664,9 @@ static void memcg_create_cache_enqueue(struct mem_cgroup *memcg,
  if (!css_tryget(&memcg->css))
   return;

+ memcg_stop_kmem_account();
  cw = kmalloc(sizeof(struct create_work), GFP_NOWAIT);
+ memcg_resume_kmem_account();
  if (cw == NULL) {
   css_put(&memcg->css);
   return;
@@ -681,6 +701,9 @@ struct kmem_cache *__mem_cgroup_get_kmem_cache(struct
kmem_cache *cachep,
  int idx;
  struct task_struct *p;

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

@@ -713,6 +736,9 @@ bool __mem_cgroup_new_kmem_page(gfp_t gfp, void *_handle, int
order)
  struct task_struct *p;

  handle = NULL;
+ if (current->memcg_kmem_skip_account)
+  return true;
+
```

```
  rcu_read_lock();
  p = rcu_dereference(current->mm->owner);
  memcg = mem_cgroup_from_task(p);
--
1.7.10.2
```

Subject: [PATCH v4 18/25] mm: Allocate kernel pages to the right memcg
Posted by Glauber Costa on Mon, 18 Jun 2012 10:28:11 GMT
View Forum Message <> Reply to Message

This patch builds on the suggestion previously given by Cristoph, with one
major difference: it still keeps the cache dispatcher and the cache duplicates.
But its internals are completely different.

I no longer mess with the cache cores when pages are allocated. (except for
destruction, that happens a bit later, but that's quite simple). All of that
is done by the page allocator, by recognizing the __GFP_SLABMEMCG flag.

The catch here is that 99% of the time, the task doing the dispatch will be
the same allocating the page. It doesn't hold only when tasks are moving around.
But that's an acceptable price to pay, at least for me. Moving around won't break,
it will at the most put us on a state where a cache has a page that is acconted
to a different cgroup. Or, if that cgroups is destroyed, not accounted to anyone.
If that ever hurts anyone, this is solvable by a reaper, or by a full cache scan
when the task moves.

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
CC: Michal Hocko <mhocko@suse.cz>
CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Johannes Weiner <hannes@cmpxchg.org>
CC: Suleiman Souhlal <suleiman@google.com>

```
 include/linux/page-flags.h |    2 +-
 include/linux/slub_def.h   |   18 ++++++++++++-----
 mm/memcontrol.c            |    2 ++
 mm/page_alloc.c            |   13 ++++++++++-
 mm/slab.c                  |    4 ++++
 mm/slub.c                  |    1 +
 6 files changed, 33 insertions(+), 7 deletions(-)
```

diff --git a/include/linux/page-flags.h b/include/linux/page-flags.h
index c88d2a9..9b065d7 100644
--- a/include/linux/page-flags.h
+++ b/include/linux/page-flags.h
@@ -201,7 +201,7 @@ PAGEFLAG(Dirty, dirty) TESTSCFLAG(Dirty, dirty)

```
__CLEARPAGEFLAG(Dirty, dirty)
 PAGEFLAG(LRU, lru) __CLEARPAGEFLAG(LRU, lru)
 PAGEFLAG(Active, active) __CLEARPAGEFLAG(Active, active)
  TESTCLEARFLAG(Active, active)
-__PAGEFLAG(Slab, slab)
+__PAGEFLAG(Slab, slab) __TESTCLEARFLAG(Slab, slab)
 PAGEFLAG(Checked, checked)  /* Used by some filesystems */
 PAGEFLAG(Pinned, pinned) TESTSCFLAG(Pinned, pinned) /* Xen */
 PAGEFLAG(SavePinned, savepinned);   /* Xen */
diff --git a/include/linux/slub_def.h b/include/linux/slub_def.h
index 7637f3b..7183596 100644
--- a/include/linux/slub_def.h
+++ b/include/linux/slub_def.h
@@ -13,6 +13,8 @@
 #include <linux/kobject.h>

 #include <linux/kmemleak.h>
+#include <linux/memcontrol.h>
+#include <linux/mm.h>

 enum stat_item {
  ALLOC_FASTPATH,  /* Allocation from cpu slab */
@@ -209,14 +211,14 @@ static __always_inline int kmalloc_index(size_t size)
  * This ought to end up with a global pointer to the right cache
  * in kmalloc_caches.
  */
-static __always_inline struct kmem_cache *kmalloc_slab(size_t size)
+static __always_inline struct kmem_cache *kmalloc_slab(gfp_t flags, size_t size)
 {
  int index = kmalloc_index(size);

  if (index == 0)
   return NULL;

- return kmalloc_caches[index];
+ return mem_cgroup_get_kmem_cache(kmalloc_caches[index], flags);
 }

 void *kmem_cache_alloc(struct kmem_cache *, gfp_t);
@@ -225,7 +227,13 @@ void *__kmalloc(size_t size, gfp_t flags);
 static __always_inline void *
 kmalloc_order(size_t size, gfp_t flags, unsigned int order)
 {
- void *ret = (void *) __get_free_pages(flags | __GFP_COMP, order);
+ void *ret;
+
+ flags = __GFP_COMP;
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
```

```
+ flags |= __GFP_SLABMEMCG;
+#endif
+ ret = (void *) __get_free_pages(flags, order);
  kmemleak_alloc(ret, size, 1, flags);
  return ret;
 }
@@ -274,7 +282,7 @@ static __always_inline void *kmalloc(size_t size, gfp_t flags)
    return kmalloc_large(size, flags);

   if (!(flags & SLUB_DMA)) {
-   struct kmem_cache *s = kmalloc_slab(size);
+   struct kmem_cache *s = kmalloc_slab(flags, size);

    if (!s)
     return ZERO_SIZE_PTR;
@@ -307,7 +315,7 @@ static __always_inline void *kmalloc_node(size_t size, gfp_t flags, int
node)
 {
  if (__builtin_constant_p(size) &&
   size <= SLUB_MAX_SIZE && !(flags & SLUB_DMA)) {
-   struct kmem_cache *s = kmalloc_slab(size);
+   struct kmem_cache *s = kmalloc_slab(flags, size);

   if (!s)
    return ZERO_SIZE_PTR;
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 233d3bc..b47ab87 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -524,6 +524,8 @@ static struct kmem_cache *kmem_cache_dup(struct mem_cgroup
*memcg,

  new = kmem_cache_create_memcg(memcg, name, s->object_size, s->align,
         (s->flags & ~SLAB_PANIC), s->ctor, s);
+ if (new)
+  new->allocflags |= __GFP_SLABMEMCG;

  kfree(name);
  return new;
diff --git a/mm/page_alloc.c b/mm/page_alloc.c
index a884a9c..b4322b7 100644
--- a/mm/page_alloc.c
+++ b/mm/page_alloc.c
@@ -2425,6 +2425,7 @@ __alloc_pages_nodemask(gfp_t gfp_mask, unsigned int order,
  struct page *page = NULL;
  int migratetype = allocflags_to_migratetype(gfp_mask);
  unsigned int cpuset_mems_cookie;
+ void *handle = NULL;
```

```
  gfp_mask &= gfp_allowed_mask;

@@ -2436,6 +2437,13 @@ __alloc_pages_nodemask(gfp_t gfp_mask, unsigned int order,
   return NULL;

  /*
+  * Will only have any effect when __GFP_SLABMEMCG is set.
+  * This is verified in the (always inline) callee
+  */
+ if (!mem_cgroup_new_kmem_page(gfp_mask, &handle, order))
+  return NULL;
+
+ /*
   * Check the zones suitable for the gfp_mask contain at least one
   * valid zone. It's possible to have an empty zonelist as a result
   * of GFP_THISNODE and a memoryless node
@@ -2474,6 +2482,8 @@ out:
  if (unlikely(!put_mems_allowed(cpuset_mems_cookie) && !page))
   goto retry_cpuset;

+ mem_cgroup_commit_kmem_page(page, handle, order);
+
  return page;
 }
 EXPORT_SYMBOL(__alloc_pages_nodemask);
@@ -2507,7 +2517,8 @@ EXPORT_SYMBOL(get_zeroed_page);
 void __free_pages(struct page *page, unsigned int order)
 {
  if (put_page_testzero(page)) {
-  __ClearPageSlab(page);
+  if (__TestClearPageSlab(page))
+   mem_cgroup_free_kmem_page(page, order);
   if (order == 0)
    free_hot_cold_page(page, 0);
   else
diff --git a/mm/slab.c b/mm/slab.c
index e537406..3da5210 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -3313,6 +3313,8 @@ __cache_alloc_node(struct kmem_cache *cachep, gfp_t flags, int
nodeid,
  if (slab_should_failslab(cachep, flags))
   return NULL;

+ cachep = mem_cgroup_get_kmem_cache(cachep, flags);
+
  cache_alloc_debugcheck_before(cachep, flags);
```

```
  local_irq_save(save_flags);

@@ -3398,6 +3400,8 @@ __cache_alloc(struct kmem_cache *cachep, gfp_t flags, void *caller)
  if (slab_should_failslab(cachep, flags))
   return NULL;

+ cachep = mem_cgroup_get_kmem_cache(cachep, flags);
+
  cache_alloc_debugcheck_before(cachep, flags);
  local_irq_save(save_flags);
  objp = __do_cache_alloc(cachep, flags);
diff --git a/mm/slub.c b/mm/slub.c
index 69c5677..77944e2 100644
--- a/mm/slub.c
+++ b/mm/slub.c
@@ -2304,6 +2304,7 @@ static __always_inline void *slab_alloc(struct kmem_cache *s,
  if (slab_pre_alloc_hook(s, gfpflags))
   return NULL;

+ s = mem_cgroup_get_kmem_cache(s, gfpflags);
 redo:

  /*
--
1.7.10.2
```

Subject: [PATCH v4 19/25] memcg: disable kmem code when not in use.
Posted by Glauber Costa on Mon, 18 Jun 2012 10:28:12 GMT
View Forum Message <> Reply to Message

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

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

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

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

```
CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Johannes Weiner <hannes@cmpxchg.org>
CC: Suleiman Souhlal <suleiman@google.com>
---
 include/linux/memcontrol.h |    5 ++++-
 mm/memcontrol.c            |   22 ++++++++++++++++++++-
 2 files changed, 25 insertions(+), 2 deletions(-)

diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
index 27a3f16..47ccd80 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;
@@ -451,7 +452,6 @@ bool __mem_cgroup_new_kmem_page(gfp_t gfp, void *handle, int
order);
 void __mem_cgroup_commit_kmem_page(struct page *page, void *handle, int order);
 void __mem_cgroup_free_kmem_page(struct page *page, int order);

-#define mem_cgroup_kmem_on 1
 struct kmem_cache *
 __mem_cgroup_get_kmem_cache(struct kmem_cache *cachep, gfp_t gfp);

@@ -459,6 +459,9 @@ static inline bool has_memcg_flag(gfp_t gfp)
 {
  return gfp & __GFP_SLABMEMCG;
 }
+
+extern struct static_key mem_cgroup_kmem_enabled_key;
+#define mem_cgroup_kmem_on static_key_false(&mem_cgroup_kmem_enabled_key)
 #else
 static inline void mem_cgroup_register_cache(struct mem_cgroup *memcg,
          struct kmem_cache *s)
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index b47ab87..5295ab6 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -422,6 +422,10 @@ static void mem_cgroup_put(struct mem_cgroup *memcg);
 #include <net/sock.h>
 #include <net/ip.h>

+struct static_key mem_cgroup_kmem_enabled_key;
```

```
+/* so modules can inline the checks */
+EXPORT_SYMBOL(mem_cgroup_kmem_enabled_key);
+
 static bool mem_cgroup_is_root(struct mem_cgroup *memcg);
 static int memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t gfp, s64 delta);
 static void memcg_uncharge_kmem(struct mem_cgroup *memcg, s64 delta);
@@ -468,6 +472,12 @@ void sock_release_memcg(struct sock *sk)
  }
 }

+static void disarm_static_keys(struct mem_cgroup *memcg)
+{
+ if (memcg->kmem_accounted)
+  static_key_slow_dec(&mem_cgroup_kmem_enabled_key);
+}
+
 #ifdef CONFIG_INET
 struct cg_proto *tcp_proto_cgroup(struct mem_cgroup *memcg)
 {
@@ -831,6 +841,10 @@ static void memcg_slab_init(struct mem_cgroup *memcg)
  for (i = 0; i < MAX_KMEM_CACHE_TYPES; i++)
   memcg->slabs[i] = NULL;
 }
+#else
+static inline void disarm_static_keys(struct mem_cgroup *memcg)
+{
+}
 #endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */

 static void drain_all_stock_async(struct mem_cgroup *memcg);
@@ -4344,8 +4358,13 @@ static int mem_cgroup_write(struct cgroup *cont, struct cftype *cft,
     *
     * But it is not worth the trouble
     */
-   if (!memcg->kmem_accounted && val != RESOURCE_MAX)
+   mutex_lock(&set_limit_mutex);
+   if (!memcg->kmem_accounted && val != RESOURCE_MAX
+       && !memcg->kmem_accounted) {
+    static_key_slow_inc(&mem_cgroup_kmem_enabled_key);
     memcg->kmem_accounted = true;
+   }
+   mutex_unlock(&set_limit_mutex);
   }
 #endif
   else
@@ -5285,6 +5304,7 @@ static void free_work(struct work_struct *work)
  int size = sizeof(struct mem_cgroup);
```

```
  memcg = container_of(work, struct mem_cgroup, work_freeing);
+ disarm_static_keys(memcg);
  if (size < PAGE_SIZE)
   kfree(memcg);
  else
--
1.7.10.2
```

Subject: [PATCH v4 20/25] memcg: destroy memcg caches
Posted by Glauber Costa on Mon, 18 Jun 2012 10:28:13 GMT
View Forum Message <> Reply to Message

This patch implements destruction of memcg caches. Right now,
only caches where our reference counter is the last remaining are
deleted. If there are any other reference counters around, we just
leave the caches lying around until they go away.

When that happen, a destruction function is called from the cache
code. Caches are only destroyed in process context, so we queue them
up for later processing in the general case.

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
CC: Michal Hocko <mhocko@suse.cz>
CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Johannes Weiner <hannes@cmpxchg.org>
CC: Suleiman Souhlal <suleiman@google.com>

```
 include/linux/memcontrol.h |    2 +
 include/linux/slab.h       |    4 +-
 mm/memcontrol.c            |   95 ++++++++++++++++++++++++++++++++++++++++++-
 mm/slab.c                  |    4 ++
 mm/slab.h                  |   24 ++++++++++
 mm/slub.c                  |    6 ++-
 6 files changed, 131 insertions(+), 4 deletions(-)

diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
index 47ccd80..8ebbcc9 100644
--- a/include/linux/memcontrol.h
+++ b/include/linux/memcontrol.h
@@ -462,6 +462,8 @@ static inline bool has_memcg_flag(gfp_t gfp)

 extern struct static_key mem_cgroup_kmem_enabled_key;
 #define mem_cgroup_kmem_on static_key_false(&mem_cgroup_kmem_enabled_key)
+
+void mem_cgroup_destroy_cache(struct kmem_cache *cachep);
```

```
 #else
 static inline void mem_cgroup_register_cache(struct mem_cgroup *memcg,
          struct kmem_cache *s)
diff --git a/include/linux/slab.h b/include/linux/slab.h
index d2d2fad..fb4fb7b 100644
--- a/include/linux/slab.h
+++ b/include/linux/slab.h
@@ -186,7 +186,9 @@ struct mem_cgroup_cache_params {
  struct mem_cgroup *memcg;
  struct kmem_cache *parent;
  int id;
- atomic_t refcnt;
+ bool dead;
+ atomic_t nr_pages;
+ struct list_head destroyed_list; /* Used when deleting memcg cache */
 };
 #endif

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 5295ab6..e0b79f0 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -476,6 +476,11 @@ static void disarm_static_keys(struct mem_cgroup *memcg)
 {
  if (memcg->kmem_accounted)
   static_key_slow_dec(&mem_cgroup_kmem_enabled_key);
+ /*
+  * This check can't live in kmem destruction function,
+  * since the charges will outlive the cgroup
+  */
+ BUG_ON(res_counter_read_u64(&memcg->kmem, RES_USAGE) != 0);
 }

 #ifdef CONFIG_INET
@@ -555,6 +560,8 @@ void mem_cgroup_register_cache(struct mem_cgroup *memcg,
 {
  int id = -1;

+ INIT_LIST_HEAD(&cachep->memcg_params.destroyed_list);
+
  if (!memcg)
   id = ida_simple_get(&cache_types, 0, MAX_KMEM_CACHE_TYPES,
         GFP_KERNEL);
@@ -595,7 +602,7 @@ static struct kmem_cache *memcg_create_kmem_cache(struct
mem_cgroup *memcg,
  mem_cgroup_get(memcg);
  memcg->slabs[idx] = new_cachep;
  new_cachep->memcg_params.memcg = memcg;
```

```
- atomic_set(&new_cachep->memcg_params.refcnt, 1);
+ atomic_set(&new_cachep->memcg_params.nr_pages , 0);
 out:
  mutex_unlock(&memcg_cache_mutex);
  return new_cachep;
@@ -610,6 +617,55 @@ struct create_work {
 /* Use a single spinlock for destruction and creation, not a frequent op */
 static DEFINE_SPINLOCK(cache_queue_lock);
 static LIST_HEAD(create_queue);
+static LIST_HEAD(destroyed_caches);
+
+static void kmem_cache_destroy_work_func(struct work_struct *w)
+{
+ struct kmem_cache *cachep;
+ struct mem_cgroup_cache_params *p, *tmp;
+ unsigned long flags;
+ LIST_HEAD(del_unlocked);
+
+ spin_lock_irqsave(&cache_queue_lock, flags);
+ list_for_each_entry_safe(p, tmp, &destroyed_caches, destroyed_list) {
+  cachep = container_of(p, struct kmem_cache, memcg_params);
+  list_move(&cachep->memcg_params.destroyed_list, &del_unlocked);
+ }
+ spin_unlock_irqrestore(&cache_queue_lock, flags);
+
+ list_for_each_entry_safe(p, tmp, &del_unlocked, destroyed_list) {
+  cachep = container_of(p, struct kmem_cache, memcg_params);
+  list_del(&cachep->memcg_params.destroyed_list);
+  if (!atomic_read(&cachep->memcg_params.nr_pages)) {
+   mem_cgroup_put(cachep->memcg_params.memcg);
+   kmem_cache_destroy(cachep);
+  }
+ }
+}
+static DECLARE_WORK(kmem_cache_destroy_work, kmem_cache_destroy_work_func);
+
+static void __mem_cgroup_destroy_cache(struct kmem_cache *cachep)
+{
+ BUG_ON(cachep->memcg_params.id != -1);
+ list_add(&cachep->memcg_params.destroyed_list, &destroyed_caches);
+}
+
+void mem_cgroup_destroy_cache(struct kmem_cache *cachep)
+{
+ unsigned long flags;
+
+ if (!cachep->memcg_params.dead)
+  return;
```

```
+ /*
+  * We have to defer the actual destroying to a workqueue, because
+  * we might currently be in a context that cannot sleep.
+  */
+ spin_lock_irqsave(&cache_queue_lock, flags);
+ __mem_cgroup_destroy_cache(cachep);
+ spin_unlock_irqrestore(&cache_queue_lock, flags);
+
+ schedule_work(&kmem_cache_destroy_work);
+}

 /*
  * Flush the queue of kmem_caches to create, because we're creating a cgroup.
@@ -631,6 +687,33 @@ void mem_cgroup_flush_cache_create_queue(void)
  spin_unlock_irqrestore(&cache_queue_lock, flags);
 }

+static void mem_cgroup_destroy_all_caches(struct mem_cgroup *memcg)
+{
+ struct kmem_cache *cachep;
+ unsigned long flags;
+ int i;
+
+ /*
+  * pre_destroy() gets called with no tasks in the cgroup.
+  * this means that after flushing the create queue, no more caches
+  * will appear
+  */
+ mem_cgroup_flush_cache_create_queue();
+
+ spin_lock_irqsave(&cache_queue_lock, flags);
+ for (i = 0; i < MAX_KMEM_CACHE_TYPES; i++) {
+  cachep = memcg->slabs[i];
+  if (!cachep)
+   continue;
+
+  cachep->memcg_params.dead = true;
+  __mem_cgroup_destroy_cache(cachep);
+ }
+ spin_unlock_irqrestore(&cache_queue_lock, flags);
+
+ schedule_work(&kmem_cache_destroy_work);
+}
+
 static void memcg_create_cache_work_func(struct work_struct *w)
 {
  struct create_work *cw, *tmp;
@@ -845,6 +928,10 @@ static void memcg_slab_init(struct mem_cgroup *memcg)
```

```
 static inline void disarm_static_keys(struct mem_cgroup *memcg)
 {
 }
+
+static inline void mem_cgroup_destroy_all_caches(struct mem_cgroup *memcg)
+{
+}
 #endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */

 static void drain_all_stock_async(struct mem_cgroup *memcg);
@@ -4118,6 +4205,7 @@ 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);

@@ -4157,8 +4245,10 @@ move_account:
   if (ret == -ENOMEM)
    goto try_to_free;
   cond_resched();
+  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;
@@ -5511,6 +5601,7 @@ static int mem_cgroup_pre_destroy(struct cgroup *cont)
 {
  struct mem_cgroup *memcg = mem_cgroup_from_cont(cont);

+ mem_cgroup_destroy_all_caches(memcg);
  return mem_cgroup_force_empty(memcg, false);
 }

diff --git a/mm/slab.c b/mm/slab.c
index 3da5210..4951c81 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -1766,6 +1766,8 @@ static void *kmem_getpages(struct kmem_cache *cachep, gfp_t flags,
int nodeid)
  for (i = 0; i < nr_pages; i++)
   __SetPageSlab(page + i);
```

```
+ mem_cgroup_bind_pages(cachep, cachep->gfporder);
+
  if (kmemcheck_enabled && !(cachep->flags & SLAB_NOTRACK)) {
   kmemcheck_alloc_shadow(page, cachep->gfporder, flags, nodeid);

@@ -1795,6 +1797,8 @@ static void kmem_freepages(struct kmem_cache *cachep, void *addr)
  else
   sub_zone_page_state(page_zone(page),
     NR_SLAB_UNRECLAIMABLE, nr_freed);
+
+ mem_cgroup_release_pages(cachep, cachep->gfporder);
  if (current->reclaim_state)
   current->reclaim_state->reclaimed_slab += nr_freed;
  free_pages((unsigned long)addr, cachep->gfporder);
diff --git a/mm/slab.h b/mm/slab.h
index 0a3e712..0b0cab4 100644
--- a/mm/slab.h
+++ b/mm/slab.h
@@ -1,5 +1,6 @@
 #ifndef MM_SLAB_H
 #define MM_SLAB_H
+#include <linux/memcontrol.h>
 /*
  * Internal slab definitions
  */
@@ -73,4 +74,27 @@ static inline bool slab_is_parent(struct kmem_cache *s,
  return false;
 #endif
 }
+
+#if defined(CONFIG_CGROUP_MEM_RES_CTLR_KMEM)
+static inline void mem_cgroup_bind_pages(struct kmem_cache *s, int order)
+{
+ if (s->memcg_params.id == -1)
+  atomic_add(1 << order, &s->memcg_params.nr_pages);
+}
+
+static inline void mem_cgroup_release_pages(struct kmem_cache *s, int order)
+{
+ if (s->memcg_params.id != -1)
+  return;
+ if (atomic_sub_and_test((1 << order), &s->memcg_params.nr_pages))
+  mem_cgroup_destroy_cache(s);
+}
+#else
+static inline void mem_cgroup_bind_pages(struct kmem_cache *s, int order)
+{
+}
```

```
+static inline void mem_cgroup_release_pages(struct kmem_cache *s, int order)
+{
+}
+#endif
 #endif
diff --git a/mm/slub.c b/mm/slub.c
index 77944e2..d5b91f4 100644
--- a/mm/slub.c
+++ b/mm/slub.c
@@ -1345,6 +1345,7 @@ static struct page *new_slab(struct kmem_cache *s, gfp_t flags, int
node)
  void *start;
  void *last;
  void *p;
+ int order;

  BUG_ON(flags & GFP_SLAB_BUG_MASK);

@@ -1353,14 +1354,16 @@ static struct page *new_slab(struct kmem_cache *s, gfp_t flags, int
node)
  if (!page)
   goto out;

+ order = compound_order(page);
  inc_slabs_node(s, page_to_nid(page), page->objects);
+ mem_cgroup_bind_pages(s, order);
  page->slab = s;
  page->flags |= 1 << PG_slab;

  start = page_address(page);

  if (unlikely(s->flags & SLAB_POISON))
-  memset(start, POISON_INUSE, PAGE_SIZE << compound_order(page));
+  memset(start, POISON_INUSE, PAGE_SIZE << order);

  last = start;
  for_each_object(p, s, start, page->objects) {
@@ -1399,6 +1402,7 @@ static void __free_slab(struct kmem_cache *s, struct page *page)
   NR_SLAB_RECLAIMABLE : NR_SLAB_UNRECLAIMABLE,
   -pages);

+ mem_cgroup_release_pages(s, order);
  reset_page_mapcount(page);
  if (current->reclaim_state)
   current->reclaim_state->reclaimed_slab += pages;
--
1.7.10.2
```

Subject: [PATCH v4 21/25] Track all the memcg children of a kmem_cache.
Posted by Glauber Costa on Mon, 18 Jun 2012 10:28:14 GMT
View Forum Message <> Reply to Message

This enables us to remove all the children of a kmem_cache being
destroyed, if for example the kernel module it's being used in
gets unloaded. Otherwise, the children will still point to the
destroyed parent.

Signed-off-by: Suleiman Souhlal <suleiman@google.com>
Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
CC: Michal Hocko <mhocko@suse.cz>
CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Johannes Weiner <hannes@cmpxchg.org>
---
 include/linux/memcontrol.h | 1 +
 include/linux/slab.h | 1 +
 mm/memcontrol.c | 16 ++++++++++++++-
 mm/slab_common.c | 31 +++++++++++++++++++++++++++++++
 4 files changed, 48 insertions(+), 1 deletion(-)

diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
index 8ebbcc9..8884a4e 100644
--- a/include/linux/memcontrol.h
+++ b/include/linux/memcontrol.h
@@ -464,6 +464,7 @@ extern struct static_key mem_cgroup_kmem_enabled_key;
 #define mem_cgroup_kmem_on static_key_false(&mem_cgroup_kmem_enabled_key)

 void mem_cgroup_destroy_cache(struct kmem_cache *cachep);
+void mem_cgroup_remove_child_kmem_cache(struct kmem_cache *cachep, int id);
 #else
 static inline void mem_cgroup_register_cache(struct mem_cgroup *memcg,
          struct kmem_cache *s)
diff --git a/include/linux/slab.h b/include/linux/slab.h
index fb4fb7b..155d19f 100644
--- a/include/linux/slab.h
+++ b/include/linux/slab.h
@@ -189,6 +189,7 @@ struct mem_cgroup_cache_params {
  bool dead;
  atomic_t nr_pages;
  struct list_head destroyed_list; /* Used when deleting memcg cache */
+ struct list_head sibling_list;
 };
 #endif

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index e0b79f0..e32b53e 100644

```
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -539,8 +539,11 @@ static struct kmem_cache *kmem_cache_dup(struct mem_cgroup
*memcg,

  new = kmem_cache_create_memcg(memcg, name, s->object_size, s->align,
          (s->flags & ~SLAB_PANIC), s->ctor, s);
- if (new)
+ if (new) {
   new->allocflags |= __GFP_SLABMEMCG;
+  list_add(&new->memcg_params.sibling_list,
+    &s->memcg_params.sibling_list);
+ }

  kfree(name);
  return new;
@@ -561,6 +564,7 @@ void mem_cgroup_register_cache(struct mem_cgroup *memcg,
  int id = -1;

  INIT_LIST_HEAD(&cachep->memcg_params.destroyed_list);
+ INIT_LIST_HEAD(&cachep->memcg_params.sibling_list);

  if (!memcg)
   id = ida_simple_get(&cache_types, 0, MAX_KMEM_CACHE_TYPES,
@@ -573,6 +577,9 @@ void mem_cgroup_release_cache(struct kmem_cache *cachep)
  mem_cgroup_flush_cache_create_queue();
  if (cachep->memcg_params.id != -1)
   ida_simple_remove(&cache_types, cachep->memcg_params.id);
+ else
+  list_del(&cachep->memcg_params.sibling_list);
+
 }

 static struct kmem_cache *memcg_create_kmem_cache(struct mem_cgroup *memcg,
@@ -776,6 +783,13 @@ static void memcg_create_cache_enqueue(struct mem_cgroup
*memcg,
  schedule_work(&memcg_create_cache_work);
 }

+void mem_cgroup_remove_child_kmem_cache(struct kmem_cache *cachep, int id)
+{
+ mutex_lock(&memcg_cache_mutex);
+ cachep->memcg_params.memcg->slabs[id] = NULL;
+ mutex_unlock(&memcg_cache_mutex);
+}
+
 /*
  * Return the kmem_cache we're supposed to use for a slab allocation.
```

```
  * We try to use the current memcg's version of the cache.
diff --git a/mm/slab_common.c b/mm/slab_common.c
index 619d365..b424b28 100644
--- a/mm/slab_common.c
+++ b/mm/slab_common.c
@@ -190,8 +190,39 @@ struct kmem_cache *kmem_cache_create(const char *name, size_t
size, size_t align
 }
 EXPORT_SYMBOL(kmem_cache_create);

+static void kmem_cache_destroy_memcg_children(struct kmem_cache *s)
+{
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ struct kmem_cache *c;
+ struct mem_cgroup_cache_params *p, *tmp;
+ int id = s->memcg_params.id;
+
+ if (id == -1)
+  return;
+
+ mutex_lock(&slab_mutex);
+ list_for_each_entry_safe(p, tmp,
+      &s->memcg_params.sibling_list, sibling_list) {
+  c = container_of(p, struct kmem_cache, memcg_params);
+  if (WARN_ON(c == s))
+   continue;
+
+  mutex_unlock(&slab_mutex);
+  BUG_ON(c->memcg_params.id != -1);
+  mem_cgroup_remove_child_kmem_cache(c, id);
+  kmem_cache_destroy(c);
+  mutex_lock(&slab_mutex);
+ }
+ mutex_unlock(&slab_mutex);
+#endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
+}
+
 void kmem_cache_destroy(struct kmem_cache *s)
 {
+
+ /* Destroy all the children caches if we aren't a memcg cache */
+ kmem_cache_destroy_memcg_children(s);
+
  get_online_cpus();
  mutex_lock(&slab_mutex);
  list_del(&s->list);
--
1.7.10.2
```

Subject: [PATCH v4 22/25] slab: slab-specific propagation changes.
Posted by Glauber Costa on Mon, 18 Jun 2012 10:28:15 GMT
View Forum Message <> Reply to Message

When a parent cache does tune_cpucache, we need to propagate that to the
children as well. For that, we unfortunately need to tap into the slab core.

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
CC: Michal Hocko <mhocko@suse.cz>
CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Johannes Weiner <hannes@cmpxchg.org>
CC: Suleiman Souhlal <suleiman@google.com>
---
 mm/slab.c | 28 ++++++++++++++++++++++++++-
 mm/slab_common.c | 1 +
 2 files changed, 28 insertions(+), 1 deletion(-)

diff --git a/mm/slab.c b/mm/slab.c
index 4951c81..c280dc6 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -3874,7 +3874,7 @@ static void do_ccupdate_local(void *info)
 }

 /* Always called with the slab_mutex held */
-static int do_tune_cpucache(struct kmem_cache *cachep, int limit,
+static int __do_tune_cpucache(struct kmem_cache *cachep, int limit,
     int batchcount, int shared, gfp_t gfp)
 {
  struct ccupdate_struct *new;
@@ -3917,6 +3917,32 @@ static int do_tune_cpucache(struct kmem_cache *cachep, int limit,
  return alloc_kmemlist(cachep, gfp);
 }

+static int do_tune_cpucache(struct kmem_cache *cachep, int limit,
+   int batchcount, int shared, gfp_t gfp)
+{
+ int ret;
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ struct kmem_cache *c;
+ struct mem_cgroup_cache_params *p;
+#endif
+
+ ret = __do_tune_cpucache(cachep, limit, batchcount, shared, gfp);
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ if (slab_state < FULL)
+ return ret;

```
+
+ if ((ret < 0) || (cachep->memcg_params.id == -1))
+  return ret;
+
+ list_for_each_entry(p, &cachep->memcg_params.sibling_list, sibling_list) {
+  c = container_of(p, struct kmem_cache, memcg_params);
+  /* return value determined by the parent cache only */
+  __do_tune_cpucache(c, limit, batchcount, shared, gfp);
+ }
+#endif
+ return ret;
+}
+
 /* Called with slab_mutex held always */
 static int enable_cpucache(struct kmem_cache *cachep, gfp_t gfp)
 {
diff --git a/mm/slab_common.c b/mm/slab_common.c
index b424b28..a8557e8 100644
--- a/mm/slab_common.c
+++ b/mm/slab_common.c
@@ -155,6 +155,7 @@ kmem_cache_create_memcg(struct mem_cgroup *memcg, const char
*name, size_t size,
  s->flags = flags;
  s->align = calculate_alignment(flags, align, size);
 #ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ s->memcg_params.id = -1; /* not registered yet */
  s->memcg_params.memcg = memcg;
  s->memcg_params.parent = parent_cache;
 #endif
--
1.7.10.2
```

Subject: [PATCH v4 23/25] memcg: propagate kmem limiting information to children
Posted by Glauber Costa on Mon, 18 Jun 2012 10:28:16 GMT

View Forum Message <> Reply to Message

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

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

* kmem limit set at A
* A and B empty taskwise
* bash in C does find /

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

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

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

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
CC: Michal Hocko <mhocko@suse.cz>
CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Johannes Weiner <hannes@cmpxchg.org>
CC: Suleiman Souhlal <suleiman@google.com>
---
 mm/memcontrol.c | 147 ++------
 1 file changed, 131 insertions(+), 16 deletions(-)

diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index e32b53e..972e83f 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -259,6 +259,9 @@ struct mem_cgroup {
   * the counter to account for kernel memory usage.
   */
  struct res_counter kmem;
+
+ struct list_head children;
+ struct list_head siblings;
  /*
   * Per cgroup active and inactive list, similar to the
   * per zone LRU lists.
@@ -274,7 +277,11 @@ struct mem_cgroup {
   * Should the accounting and control be hierarchical, per subtree?
   */
  bool use_hierarchy;
- bool kmem_accounted;
+ /*
+ * bit0: accounted by this cgroup
+ * bit1: accounted by a parent.
+ */
+ volatile unsigned long kmem_accounted;

```
  bool  oom_lock;
  atomic_t under_oom;
@@ -332,6 +339,9 @@ struct mem_cgroup {
 #endif
 };

+#define KMEM_ACCOUNTED_THIS 0
+#define KMEM_ACCOUNTED_PARENT 1
+
 int memcg_css_id(struct mem_cgroup *memcg)
 {
  return css_id(&memcg->css);
@@ -474,7 +484,7 @@ void sock_release_memcg(struct sock *sk)

 static void disarm_static_keys(struct mem_cgroup *memcg)
 {
- if (memcg->kmem_accounted)
+ if (test_bit(KMEM_ACCOUNTED_THIS, &memcg->kmem_accounted))
    static_key_slow_dec(&mem_cgroup_kmem_enabled_key);
   /*
    * This check can't live in kmem destruction function,
@@ -4418,6 +4428,110 @@ 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);
 }
+
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+typedef int (*memcg_visitor)(struct mem_cgroup*, void *);
+
+/*
+ * This is mostly "inspired" by the code in sched/core.c. I decided to copy it,
+ * instead of factoring it, because of all the typing issues we'd run into.
+ * In particular, grabbing the parent is very different for memcg, because we
+ * may or may not have hierarchy, while cpu cgroups always do. That would lead
+ * to either indirect calls - this is not a fast path for us, but can be for
+ * the scheduler - or a big and ugly macro.
+ *
+ * If we ever get rid of hierarchy, we could iterate over struct cgroup, and
+ * then it would cease to be a problem.
+ */
+int walk_tree_from(struct mem_cgroup *from,
+     memcg_visitor down, memcg_visitor up, void *data)
+{
+ struct mem_cgroup *parent, *child;
+ int ret;
+
```

```
+
+ parent = from;
+down:
+ ret = (*down)(parent, data);
+ if (ret)
+  goto out;
+
+ list_for_each_entry_rcu(child, &parent->children, siblings) {
+  parent = child;
+  goto down;
+
+up:
+  continue;
+ }
+ ret = (*up)(parent, data);
+ if (ret || parent == from)
+  goto out;
+
+ child = parent;
+ parent = parent_mem_cgroup(parent);
+ if (parent)
+  goto up;
+out:
+ return ret;
+}
+
+static int memcg_nop(struct mem_cgroup *memcg, void *data)
+{
+ return 0;
+}
+
+static int memcg_parent_account(struct mem_cgroup *memcg, void *data)
+{
+ if (memcg == data)
+  return 0;
+
+ set_bit(KMEM_ACCOUNTED_PARENT, &memcg->kmem_accounted);
+ return 0;
+}
+
+static int memcg_parent_no_account(struct mem_cgroup *memcg, void *data)
+{
+ if (memcg == data)
+  return 0;
+
+ clear_bit(KMEM_ACCOUNTED_PARENT, &memcg->kmem_accounted);
+ /*
+  * Stop propagation if we are accounted: our children should
```

```
+  * be parent-accounted
+  */
+ return test_bit(KMEM_ACCOUNTED_THIS, &memcg->kmem_accounted);
+}
+
+static void mem_cgroup_update_kmem_limit(struct mem_cgroup *memcg, u64 val)
+{
+ mutex_lock(&set_limit_mutex);
+ if (!test_and_set_bit(KMEM_ACCOUNTED_THIS, &memcg->kmem_accounted) &&
+  val != RESOURCE_MAX) {
+
+  /*
+   * Once enabled, can't be disabled. We could in theory
+   * disable it if we haven't yet created any caches, or
+   * if we can shrink them all to death.
+   *
+   * But it is not worth the trouble
+   */
+  static_key_slow_inc(&mem_cgroup_kmem_enabled_key);
+
+  rcu_read_lock();
+  walk_tree_from(memcg, memcg_parent_account, memcg_nop, memcg);
+  rcu_read_unlock();
+ } else if (test_and_clear_bit(KMEM_ACCOUNTED_THIS, &memcg->kmem_accounted)
+  && val == RESOURCE_MAX) {
+
+  rcu_read_lock();
+  walk_tree_from(memcg, memcg_parent_no_account,
+          memcg_nop, memcg);
+  rcu_read_unlock();
+ }
+
+ mutex_unlock(&set_limit_mutex);
+}
+#endif
 /*
  * The user of this function is...
  * RES_LIMIT.
@@ -4455,20 +4569,8 @@ static int mem_cgroup_write(struct cgroup *cont, struct cftype *cft,
    ret = res_counter_set_limit(&memcg->kmem, val);
    if (ret)
     break;
-   /*
-    * Once enabled, can't be disabled. We could in theory
-    * disable it if we haven't yet created any caches, or
-    * if we can shrink them all to death.
-    *
-    * But it is not worth the trouble
-    */
-   if (!test_and_set_bit(KMEM_ACCOUNTED_THIS, &memcg->kmem_accounted) &&
-       val != RESOURCE_MAX)
-    static_key_slow_inc(&mem_cgroup_kmem_enabled_key);
```

```
-    */
-   mutex_lock(&set_limit_mutex);
-   if (!memcg->kmem_accounted && val != RESOURCE_MAX
-       && !memcg->kmem_accounted) {
-    static_key_slow_inc(&mem_cgroup_kmem_enabled_key);
-    memcg->kmem_accounted = true;
-   }
-   mutex_unlock(&set_limit_mutex);
+   mem_cgroup_update_kmem_limit(memcg, val);
+   break;
   }
 #endif
   else
@@ -5527,6 +5629,8 @@ err_cleanup:

 }

+static DEFINE_MUTEX(memcg_list_mutex);
+
 static struct cgroup_subsys_state * __ref
 mem_cgroup_create(struct cgroup *cont)
 {
@@ -5542,6 +5646,7 @@ mem_cgroup_create(struct cgroup *cont)
   if (alloc_mem_cgroup_per_zone_info(memcg, node))
    goto free_out;

+ INIT_LIST_HEAD(&memcg->children);
  /* root ? */
  if (cont->parent == NULL) {
   int cpu;
@@ -5580,6 +5685,10 @@ mem_cgroup_create(struct cgroup *cont)
    * mem_cgroup(see mem_cgroup_put).
    */
   mem_cgroup_get(parent);
+
+  mutex_lock(&memcg_list_mutex);
+  list_add_rcu(&memcg->siblings, &parent->children);
+  mutex_unlock(&memcg_list_mutex);
  } else {
   res_counter_init(&memcg->res, NULL);
   res_counter_init(&memcg->memsw, NULL);
@@ -5622,9 +5731,15 @@ static int mem_cgroup_pre_destroy(struct cgroup *cont)
 static void mem_cgroup_destroy(struct cgroup *cont)
 {
  struct mem_cgroup *memcg = mem_cgroup_from_cont(cont);
+ struct mem_cgroup *parent = parent_mem_cgroup(memcg);

  kmem_cgroup_destroy(memcg);
```

```
+ mutex_lock(&memcg_list_mutex);
+ if (parent)
+  list_del_rcu(&memcg->siblings);
+ mutex_unlock(&memcg_list_mutex);
+
  mem_cgroup_put(memcg);
 }

--
1.7.10.2
```

Subject: [PATCH v4 24/25] memcg/slub: shrink dead caches
Posted by Glauber Costa on Mon, 18 Jun 2012 10:28:17 GMT
View Forum Message <> Reply to Message

In the slub allocator, when the last object of a page goes away, we
don't necessarily free it - there is not necessarily a test for empty
page in any slab_free path.

This means that when we destroy a memcg cache that happened to be empty,
those caches may take a lot of time to go away: removing the memcg
reference won't destroy them - because there are pending references,
and the empty pages will stay there, until a shrinker is called upon
for any reason.

This patch marks all memcg caches as dead. kmem_cache_shrink is called
for the ones who are not yet dead - this will force internal cache
reorganization, and then all references to empty pages will be removed.

An unlikely branch is used to make sure this case does not affect
performance in the usual slab_free path.

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
CC: Michal Hocko <mhocko@suse.cz>
CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Johannes Weiner <hannes@cmpxchg.org>
CC: Suleiman Souhlal <suleiman@google.com>

```
 include/linux/slab.h     |    3 +++
 include/linux/slub_def.h |    8 ++++++++
 mm/memcontrol.c          |   44 ++++++++++++++++++++++++++++++++++++++++++++-
 mm/slub.c                |    1 +
 4 files changed, 55 insertions(+), 1 deletion(-)
```

```
diff --git a/include/linux/slab.h b/include/linux/slab.h
index 155d19f..7e13055 100644
--- a/include/linux/slab.h
+++ b/include/linux/slab.h
@@ -182,6 +182,8 @@ unsigned int kmem_cache_size(struct kmem_cache *);
 #endif

 #ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+#include <linux/workqueue.h>
+
 struct mem_cgroup_cache_params {
  struct mem_cgroup *memcg;
  struct kmem_cache *parent;
@@ -190,6 +192,7 @@ struct mem_cgroup_cache_params {
  atomic_t nr_pages;
  struct list_head destroyed_list; /* Used when deleting memcg cache */
  struct list_head sibling_list;
+ struct work_struct cache_shrinker;
 };
 #endif

diff --git a/include/linux/slub_def.h b/include/linux/slub_def.h
index 7183596..871f82b 100644
--- a/include/linux/slub_def.h
+++ b/include/linux/slub_def.h
@@ -116,6 +116,14 @@ struct kmem_cache {
  struct kmem_cache_node *node[MAX_NUMNODES];
 };

+static inline void kmem_cache_verify_dead(struct kmem_cache *cachep)
+{
+#ifdef CONFIG_CGROUP_MEM_RES_CTLR_KMEM
+ if (unlikely(cachep->memcg_params.dead))
+  schedule_work(&cachep->memcg_params.cache_shrinker);
+#endif
+}
+
 /*
  * Kmalloc subsystem.
  */
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 972e83f..c368480 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -531,7 +531,7 @@ static char *mem_cgroup_cache_name(struct mem_cgroup *memcg,
struct kmem_cache *

  BUG_ON(dentry == NULL);
```

```
- name = kasprintf(GFP_KERNEL, "%s(%d:%s)",
+ name = kasprintf(GFP_KERNEL, "%s(%d:%s)dead",
      cachep->name, css_id(&memcg->css), dentry->d_name.name);

  return name;
@@ -592,11 +592,24 @@ void mem_cgroup_release_cache(struct kmem_cache *cachep)

 }

+static void cache_shrinker_work_func(struct work_struct *work)
+{
+ struct mem_cgroup_cache_params *params;
+ struct kmem_cache *cachep;
+
+ params = container_of(work, struct mem_cgroup_cache_params,
+         cache_shrinker);
+ cachep = container_of(params, struct kmem_cache, memcg_params);
+
+ kmem_cache_shrink(cachep);
+}
+
 static struct kmem_cache *memcg_create_kmem_cache(struct mem_cgroup *memcg,
         struct kmem_cache *cachep)
 {
  struct kmem_cache *new_cachep;
  int idx;
+ char *name;

  BUG_ON(!mem_cgroup_kmem_enabled(memcg));

@@ -616,10 +629,21 @@ static struct kmem_cache *memcg_create_kmem_cache(struct
mem_cgroup *memcg,
   goto out;
  }

+ /*
+  * Because the cache is expected to duplicate the string,
+  * we must make sure it has opportunity to copy its full
+  * name. Only now we can remove the dead part from it
+  */
+ name = (char *)new_cachep->name;
+ if (name)
+  name[strlen(name) - 4] = '\0';
+
  mem_cgroup_get(memcg);
  memcg->slabs[idx] = new_cachep;
  new_cachep->memcg_params.memcg = memcg;
```

```
  atomic_set(&new_cachep->memcg_params.nr_pages , 0);
+ INIT_WORK(&new_cachep->memcg_params.cache_shrinker,
+    cache_shrinker_work_func);
 out:
  mutex_unlock(&memcg_cache_mutex);
  return new_cachep;
@@ -642,6 +666,21 @@ static void kmem_cache_destroy_work_func(struct work_struct *w)
  struct mem_cgroup_cache_params *p, *tmp;
  unsigned long flags;
  LIST_HEAD(del_unlocked);
+ LIST_HEAD(shrinkers);
+
+ spin_lock_irqsave(&cache_queue_lock, flags);
+ list_for_each_entry_safe(p, tmp, &destroyed_caches, destroyed_list) {
+  cachep = container_of(p, struct kmem_cache, memcg_params);
+  if (atomic_read(&cachep->memcg_params.nr_pages) != 0)
+   list_move(&cachep->memcg_params.destroyed_list, &shrinkers);
+ }
+ spin_unlock_irqrestore(&cache_queue_lock, flags);
+
+ list_for_each_entry_safe(p, tmp, &shrinkers, destroyed_list) {
+  cachep = container_of(p, struct kmem_cache, memcg_params);
+  list_del(&cachep->memcg_params.destroyed_list);
+  kmem_cache_shrink(cachep);
+ }

  spin_lock_irqsave(&cache_queue_lock, flags);
  list_for_each_entry_safe(p, tmp, &destroyed_caches, destroyed_list) {
@@ -719,11 +758,14 @@ static void mem_cgroup_destroy_all_caches(struct mem_cgroup
*memcg)

  spin_lock_irqsave(&cache_queue_lock, flags);
  for (i = 0; i < MAX_KMEM_CACHE_TYPES; i++) {
+  char *name;
   cachep = memcg->slabs[i];
   if (!cachep)
    continue;

   cachep->memcg_params.dead = true;
+  name = (char *)cachep->name;
+  name[strlen(name)] = 'd';
   __mem_cgroup_destroy_cache(cachep);
  }
  spin_unlock_irqrestore(&cache_queue_lock, flags);
diff --git a/mm/slub.c b/mm/slub.c
index d5b91f4..37ac548 100644
--- a/mm/slub.c
+++ b/mm/slub.c
```

```
@@ -2593,6 +2593,7 @@ redo:
  } else
   __slab_free(s, page, x, addr);

+ kmem_cache_verify_dead(s);
 }

 void kmem_cache_free(struct kmem_cache *s, void *x)
--
1.7.10.2
```

Subject: [PATCH v4 25/25] Documentation: add documentation for slab tracker for memcg
Posted by Glauber Costa on Mon, 18 Jun 2012 10:28:18 GMT
View Forum Message <> Reply to Message

In a separate patch, to aid reviewers.

Signed-off-by: Glauber Costa <glommer@parallels.com>
CC: Christoph Lameter <cl@linux.com>
CC: Pekka Enberg <penberg@cs.helsinki.fi>
CC: Michal Hocko <mhocko@suse.cz>
CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
CC: Johannes Weiner <hannes@cmpxchg.org>
CC: Suleiman Souhlal <suleiman@google.com>
CC: Randy Dunlap <rdunlap@xenotime.net>

 Documentation/cgroups/memory.txt | 33 +++++++++++++++++++++++++++++++++
 1 file changed, 33 insertions(+)

```
diff --git a/Documentation/cgroups/memory.txt b/Documentation/cgroups/memory.txt
index 9b1067a..9ea82b5 100644
--- a/Documentation/cgroups/memory.txt
+++ b/Documentation/cgroups/memory.txt
@@ -74,6 +74,12 @@ Brief summary of control files.
  memory.kmem.tcp.limit_in_bytes  # set/show hard limit for tcp buf memory
  memory.kmem.tcp.usage_in_bytes  # show current tcp buf memory allocation

+ memory.kmem.limit_in_bytes  # set/show hard limit for general kmem memory
+ memory.kmem.usage_in_bytes  # show current general kmem memory allocation
+ memory.kmem.failcnt   # show current number of kmem limit hits
+ memory.kmem.max_usage_in_bytes  # show max kmem usage
+ memory.kmem.slabinfo   # show cgroup-specific slab usage information
+
 1. History

 The memory controller has a long history. A request for comments for the memory
```

```
@@ -270,6 +276,14 @@ cgroup may or may not be accounted.
 Currently no soft limit is implemented for kernel memory. It is future work
 to trigger slab reclaim when those limits are reached.

+Kernel memory is not accounted until it is limited. Users that want to just
+track kernel memory usage can set the limit value to a big enough value so
+the limit is guaranteed to never hit. A kernel memory limit bigger than the
+current memory limit will have this effect as well.
+
+This guarantes that this extension is backwards compatible to any previous
+memory cgroup version.
+
 2.7.1 Current Kernel Memory resources accounted

 * sockets memory pressure: some sockets protocols have memory pressure
@@ -278,6 +292,24 @@ per cgroup, instead of globally.

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

+* slab/kmalloc:
+
+When slab memory is tracked (memory.kmem.limit_in_bytes != -1ULL), both
+memory.kmem.usage_in_bytes and memory.usage_in_bytes are updated. When
+memory.kmem.limit_in_bytes is left alone, no tracking of slab caches takes
+place.
+
+Because a slab page is shared among many tasks, it is not possible to take
+any meaningful action upon task migration. Slabs created in a cgroup stay
+around until the cgroup is destructed. Information about the slabs used
+by the cgroup is displayed in the cgroup file memory.kmem.slabinfo. The format
+of this file is and should remain compatible with /proc/slabinfo.
+
+Upon cgroup destruction, slabs that holds no live references are destructed.
+Workers are fired to destroy the remaining caches as they objects are freed.
+
+Memory used by dead caches are shown in the proc file /proc/dead_slabinfo
+
 3. User Interface

 0. Configuration
@@ -286,6 +318,7 @@ a. Enable CONFIG_CGROUPS
 b. Enable CONFIG_RESOURCE_COUNTERS
 c. Enable CONFIG_CGROUP_MEM_RES_CTLR
 d. Enable CONFIG_CGROUP_MEM_RES_CTLR_SWAP (to use swap extension)
+d. Enable CONFIG_CGROUP_MEM_RES_CTLR_KMEM (to use experimental kmem extension)

 1. Prepare the cgroups (see cgroups.txt, Why are cgroups needed?)
 # mount -t tmpfs none /sys/fs/cgroup
```

--
1.7.10.2

Subject: Re: [PATCH v4 05/25] memcg: Always free struct memcg through schedule_work()
Posted by KAMEZAWA Hiroyuki on Mon, 18 Jun 2012 12:07:41 GMT
View Forum Message <> Reply to Message

(2012/06/18 19:27), Glauber Costa wrote:
> Right now we free struct memcg with kfree right after a
> rcu grace period, but defer it if we need to use vfree() to get
> rid of that memory area. We do that by need, because we need vfree
> to be called in a process context.
>
> This patch unifies this behavior, by ensuring that even kfree will
> happen in a separate thread. The goal is to have a stable place to
> call the upcoming jump label destruction function outside the realm
> of the complicated and quite far-reaching cgroup lock (that can't be
> held when calling neither the cpu_hotplug.lock nor the jump_label_mutex)
>
> Signed-off-by: Glauber Costa<glommer@parallels.com>
> CC: Tejun Heo<tj@kernel.org>
> CC: Li Zefan<lizefan@huawei.com>
> CC: Kamezawa Hiroyuki<kamezawa.hiroyu@jp.fujitsu.com>
> CC: Johannes Weiner<hannes@cmpxchg.org>
> CC: Michal Hocko<mhocko@suse.cz>

How about cut out this patch and merge first as simple cleanu up and
to reduce patch stack on your side ?

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

> ---
> mm/memcontrol.c | 24 ++++++++++++++----------
> 1 file changed, 13 insertions(+), 11 deletions(-)
>
> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
> index e3b528e..ce15be4 100644
> --- a/mm/memcontrol.c
> +++ b/mm/memcontrol.c
> @@ -245,8 +245,8 @@ struct mem_cgroup {
> */
> struct rcu_head rcu_freeing;
> /*
> - * But when using vfree(), that cannot be done at
> - * interrupt time, so we must then queue the work.
> + * We also need some space for a worker in deferred freeing.

```
> +   * By the time we call it, rcu_freeing is not longer in use.
>      */
>     struct work_struct work_freeing;
>    };
> @@ -4826,23 +4826,28 @@ out_free:
>   }
>
>   /*
> - * Helpers for freeing a vzalloc()ed mem_cgroup by RCU,
> + * 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 vfree_work(struct work_struct *work)
> +static void free_work(struct work_struct *work)
>   {
>    struct mem_cgroup *memcg;
> + int size = sizeof(struct mem_cgroup);
>
>    memcg = container_of(work, struct mem_cgroup, work_freeing);
> - vfree(memcg);
> + if (size<  PAGE_SIZE)
> +  kfree(memcg);
> + else
> +  vfree(memcg);
>   }
> -static void vfree_rcu(struct rcu_head *rcu_head)
> +
> +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, vfree_work);
> + INIT_WORK(&memcg->work_freeing, free_work);
>    schedule_work(&memcg->work_freeing);
>   }
>
> @@ -4868,10 +4873,7 @@ static void __mem_cgroup_free(struct mem_cgroup *memcg)
>     free_mem_cgroup_per_zone_info(memcg, node);
>
>    free_percpu(memcg->stat);
> - if (sizeof(struct mem_cgroup)<  PAGE_SIZE)
> -  kfree_rcu(memcg, rcu_freeing);
> - else
> -  call_rcu(&memcg->rcu_freeing, vfree_rcu);
> + call_rcu(&memcg->rcu_freeing, free_rcu);
>   }
```

>
> static void mem_cgroup_get(struct mem_cgroup *memcg)

Subject: Re: [PATCH v4 05/25] memcg: Always free struct memcg through schedule_work()
Posted by Glauber Costa on Mon, 18 Jun 2012 12:10:22 GMT
View Forum Message <> Reply to Message

On 06/18/2012 04:07 PM, Kamezawa Hiroyuki wrote:
> (2012/06/18 19:27), Glauber Costa wrote:
>> Right now we free struct memcg with kfree right after a
>> rcu grace period, but defer it if we need to use vfree() to get
>> rid of that memory area. We do that by need, because we need vfree
>> to be called in a process context.
>>
>> This patch unifies this behavior, by ensuring that even kfree will
>> happen in a separate thread. The goal is to have a stable place to
>> call the upcoming jump label destruction function outside the realm
>> of the complicated and quite far-reaching cgroup lock (that can't be
>> held when calling neither the cpu_hotplug.lock nor the jump_label_mutex)
>>
>> Signed-off-by: Glauber Costa<glommer@parallels.com>
>> CC: Tejun Heo<tj@kernel.org>
>> CC: Li Zefan<lizefan@huawei.com>
>> CC: Kamezawa Hiroyuki<kamezawa.hiroyu@jp.fujitsu.com>
>> CC: Johannes Weiner<hannes@cmpxchg.org>
>> CC: Michal Hocko<mhocko@suse.cz>
>
> How about cut out this patch and merge first as simple cleanu up and
> to reduce patch stack on your side ?
>
> Acked-by: KAMEZAWA Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>

I believe this is already in the -mm tree (from the sock memcg fixes)

But actually, my main trouble with this series here, is that I am basing
it on Pekka's tree, while some of the fixes are in -mm already.
If I'd base it on -mm I would lose some of the stuff as well.

Maybe Pekka can merge the current -mm with his tree?

So far I am happy with getting comments from people about the code, so I
did not get overly concerned about that.

Subject: Re: [PATCH v4 00/25] kmem limitation for memcg

Posted by KAMEZAWA Hiroyuki on Mon, 18 Jun 2012 12:10:38 GMT
View Forum Message <> Reply to Message

(2012/06/18 19:27), Glauber Costa wrote:
> Hello All,
>
> This is my new take for the memcg kmem accounting. This should merge
> all of the previous comments from you guys, specially concerning the big churn
> inside the allocators themselves.
>
> My focus in this new round was to keep the changes in the cache internals to
> a minimum. To do that, I relied upon two main pillars:
>
> * Cristoph's unification series, that allowed me to put must of the changes
>   in a common file. Even then, the changes are not too many, since the overal
>   level of invasiveness was decreased.
> * Accounting is done directly from the page allocator. This means some pages
>   can fail to be accounted, but that can only happen when the task calling
>   kmem_cache_alloc or kmalloc is not the same task allocating a new page.
>   This never happens in steady state operation if the tasks are kept in the
>   same memcg. Naturally, if the page ends up being accounted to a memcg that
>   is not limited (such as root memcg), that particular page will simply not
>   be accounted.
>
> The dispatcher code stays (mem_cgroup_get_kmem_cache), being the mechanism who
> guarantees that, during steady state operation, all objects allocated in a page
> will belong to the same memcg. I consider this a good compromise point between
> strict and loose accounting here.
>

2 questions.

 - Do you have performance numbers ?

 - Do you think user-memory memcg should be switched to page-allocator level accounting ?
   (it will require some study for modifying current bached-freeing and per-cpu-stock
    logics...)

Thanks,
-Kame

Subject: Re: [PATCH v4 00/25] kmem limitation for memcg
Posted by Glauber Costa on Mon, 18 Jun 2012 12:14:35 GMT
View Forum Message <> Reply to Message

On 06/18/2012 04:10 PM, Kamezawa Hiroyuki wrote:
> (2012/06/18 19:27), Glauber Costa wrote:

>> Hello All,
>>
>> This is my new take for the memcg kmem accounting. This should merge
>> all of the previous comments from you guys, specially concerning the big churn
>> inside the allocators themselves.
>>
>> My focus in this new round was to keep the changes in the cache internals to
>> a minimum. To do that, I relied upon two main pillars:
>>
>> * Cristoph's unification series, that allowed me to put must of the changes
>> in a common file. Even then, the changes are not too many, since the overal
>> level of invasiveness was decreased.
>> * Accounting is done directly from the page allocator. This means some pages
>> can fail to be accounted, but that can only happen when the task calling
>> kmem_cache_alloc or kmalloc is not the same task allocating a new page.
>> This never happens in steady state operation if the tasks are kept in the
>> same memcg. Naturally, if the page ends up being accounted to a memcg that
>> is not limited (such as root memcg), that particular page will simply not
>> be accounted.
>>
>> The dispatcher code stays (mem_cgroup_get_kmem_cache), being the mechanism who
>> guarantees that, during steady state operation, all objects allocated in a page
>> will belong to the same memcg. I consider this a good compromise point between
>> strict and loose accounting here.
>>
>
> 2 questions.
>
> - Do you have performance numbers ?

Not extensive. I've run some microbenchmarks trying to determine the
effect of my code on kmem_cache_alloc, and found it to be in the order
of 2 to 3 %. I would expect that to vanish in a workload benchmark.

>
> - Do you think user-memory memcg should be switched to page-allocator level accounting ?
> (it will require some study for modifying current bached-freeing and per-cpu-stock
> logics...)

I don't see a reason for that. My main goal by doing that was to reduce
the churn in the cache internal structures, but specially because there
is at least two of them, obeying a stable interface. The way I
understand it, memcg for user pages is already pretty well integrated to
the page allocator, so the benefit of it is questionable.

Subject: Re: [PATCH v4 17/25] skip memcg kmem allocations in specified code

regions
Posted by KAMEZAWA Hiroyuki on Mon, 18 Jun 2012 12:19:49 GMT
View Forum Message <> Reply to Message

(2012/06/18 19:28), Glauber Costa wrote:
> This patch creates a mechanism that skip memcg allocations during
> certain pieces of our core code. It basically works in the same way
> as preempt_disable()/preempt_enable(): By marking a region under
> which all allocations will be accounted to the root memcg.
>
> We need this to prevent races in early cache creation, when we
> allocate data using caches that are not necessarily created already.
>
> Signed-off-by: Glauber Costa<glommer@parallels.com>
> CC: Christoph Lameter<cl@linux.com>
> CC: Pekka Enberg<penberg@cs.helsinki.fi>
> CC: Michal Hocko<mhocko@suse.cz>
> CC: Kamezawa Hiroyuki<kamezawa.hiroyu@jp.fujitsu.com>
> CC: Johannes Weiner<hannes@cmpxchg.org>
> CC: Suleiman Souhlal<suleiman@google.com>

I'm ok with this approach.

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

Subject: Re: [PATCH v4 19/25] memcg: disable kmem code when not in use.
Posted by KAMEZAWA Hiroyuki on Mon, 18 Jun 2012 12:22:14 GMT
View Forum Message <> Reply to Message

(2012/06/18 19:28), Glauber Costa wrote:
> We can use jump labels to patch the code in or out
> when not used.
>
> Because the assignment: memcg->kmem_accounted = true
> is done after the jump labels increment, we guarantee
> that the root memcg will always be selected until
> all call sites are patched (see mem_cgroup_kmem_enabled).
> This guarantees that no mischarges are applied.
>
> Jump label decrement happens when the last reference
> count from the memcg dies. This will only happen when
> the caches are all dead.
>
> Signed-off-by: Glauber Costa<glommer@parallels.com>
> CC: Christoph Lameter<cl@linux.com>
> CC: Pekka Enberg<penberg@cs.helsinki.fi>
> CC: Michal Hocko<mhocko@suse.cz>

> CC: Kamezawa Hiroyuki<kamezawa.hiroyu@jp.fujitsu.com>
> CC: Johannes Weiner<hannes@cmpxchg.org>
> CC: Suleiman Souhlal<suleiman@google.com>
> ---
> include/linux/memcontrol.h | 5 ++++-
> mm/memcontrol.c | 22 ++++++++++++++++++++-
> 2 files changed, 25 insertions(+), 2 deletions(-)
>
> diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h
> index 27a3f16..47ccd80 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;
> @@ -451,7 +452,6 @@ bool __mem_cgroup_new_kmem_page(gfp_t gfp, void *handle, int
order);
> void __mem_cgroup_commit_kmem_page(struct page *page, void *handle, int order);
> void __mem_cgroup_free_kmem_page(struct page *page, int order);
>
> -#define mem_cgroup_kmem_on 1
> struct kmem_cache *
> __mem_cgroup_get_kmem_cache(struct kmem_cache *cachep, gfp_t gfp);
>
> @@ -459,6 +459,9 @@ static inline bool has_memcg_flag(gfp_t gfp)
> {
>   return gfp& __GFP_SLABMEMCG;
> }
> +
> +extern struct static_key mem_cgroup_kmem_enabled_key;
> +#define mem_cgroup_kmem_on static_key_false(&mem_cgroup_kmem_enabled_key)
> #else
> static inline void mem_cgroup_register_cache(struct mem_cgroup *memcg,
>   struct kmem_cache *s)
> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
> index b47ab87..5295ab6 100644
> --- a/mm/memcontrol.c
> +++ b/mm/memcontrol.c
> @@ -422,6 +422,10 @@ static void mem_cgroup_put(struct mem_cgroup *memcg);
> #include<net/sock.h>
> #include<net/ip.h>
>
> +struct static_key mem_cgroup_kmem_enabled_key;

```
> +/* so modules can inline the checks */
> +EXPORT_SYMBOL(mem_cgroup_kmem_enabled_key);
> +
>  static bool mem_cgroup_is_root(struct mem_cgroup *memcg);
>  static int memcg_charge_kmem(struct mem_cgroup *memcg, gfp_t gfp, s64 delta);
>  static void memcg_uncharge_kmem(struct mem_cgroup *memcg, s64 delta);
> @@ -468,6 +472,12 @@ void sock_release_memcg(struct sock *sk)
>   }
>  }
>
> +static void disarm_static_keys(struct mem_cgroup *memcg)
> +{
> + if (memcg->kmem_accounted)
> +  static_key_slow_dec(&mem_cgroup_kmem_enabled_key);
> +}
> +
>  #ifdef CONFIG_INET
>  struct cg_proto *tcp_proto_cgroup(struct mem_cgroup *memcg)
>  {
> @@ -831,6 +841,10 @@ static void memcg_slab_init(struct mem_cgroup *memcg)
>   for (i = 0; i<  MAX_KMEM_CACHE_TYPES; i++)
>    memcg->slabs[i] = NULL;
>  }
> +#else
> +static inline void disarm_static_keys(struct mem_cgroup *memcg)
> +{
> +}
>  #endif /* CONFIG_CGROUP_MEM_RES_CTLR_KMEM */
>
>  static void drain_all_stock_async(struct mem_cgroup *memcg);
> @@ -4344,8 +4358,13 @@ static int mem_cgroup_write(struct cgroup *cont, struct cftype *cft,
>       *
>       * But it is not worth the trouble
>       */
> -  if (!memcg->kmem_accounted&&  val != RESOURCE_MAX)
> +  mutex_lock(&set_limit_mutex);
> +  if (!memcg->kmem_accounted&&  val != RESOURCE_MAX
> +  &&  !memcg->kmem_accounted) {
```

I'm sorry why you check the value twice ?

Thanks,
-Kame

Subject: Re: [PATCH v4 19/25] memcg: disable kmem code when not in use.
Posted by Glauber Costa on Mon, 18 Jun 2012 12:26:49 GMT

>>
>> static void drain_all_stock_async(struct mem_cgroup *memcg);
>> @@ -4344,8 +4358,13 @@ static int mem_cgroup_write(struct cgroup *cont, struct cftype *cft,
>> *
>> * But it is not worth the trouble
>> */
>> - if (!memcg->kmem_accounted&& val != RESOURCE_MAX)
>> + mutex_lock(&set_limit_mutex);
>> + if (!memcg->kmem_accounted&& val != RESOURCE_MAX
>> + && !memcg->kmem_accounted) {
>
> I'm sorry why you check the value twice ?
>

Hi Kame,

For no reason, it should be removed. I never noticed this because 1)
This is the kind of thing testing will never reveal, and 2), this
actually goes away in a later patch (memcg: propagate kmem limiting
information to children)

In any case, I will update my tree here.

Thanks for spotting this

Subject: Re: [PATCH v4 23/25] memcg: propagate kmem limiting information to children
Posted by KAMEZAWA Hiroyuki on Mon, 18 Jun 2012 12:37:01 GMT

(2012/06/18 19:28), Glauber Costa wrote:
> The current memcg slab cache management fails to present satisfatory hierarchical
> behavior in the following scenario:
>
> -> /cgroups/memory/A/B/C
>
> * kmem limit set at A
> * A and B empty taskwise
> * bash in C does find /
>
> Because kmem_accounted is a boolean that was not set for C, no accounting
> would be done. This is, however, not what we expect.
>

Hmm....do we need this new routines even while we have mem_cgroup_iter() ?

Doesn't this work ?

```
 struct mem_cgroup {
  .....
  bool kmem_accounted_this;
  atomic_t kmem_accounted;
  ....
 }
```

at set limit

```
 ....set_limit(memcg) {

  if (newly accounted) {
   mem_cgroup_iter() {
    atomic_inc(&iter->kmem_accounted)
   }
  } else {
   mem_cgroup_iter() {
    atomic_dec(&iter->kmem_accounted);
   }
 }
```

hm ? Then, you can see kmem is accounted or not by atomic_read(&memcg->kmem_accounted);

Thanks,
-Kame

Subject: Re: [PATCH v4 23/25] memcg: propagate kmem limiting information to children
Posted by Glauber Costa on Mon, 18 Jun 2012 12:43:39 GMT
View Forum Message <> Reply to Message

On 06/18/2012 04:37 PM, Kamezawa Hiroyuki wrote:
> (2012/06/18 19:28), Glauber Costa wrote:
>> The current memcg slab cache management fails to present satisfatory hierarchical
>> behavior in the following scenario:
>>
>> -> /cgroups/memory/A/B/C
>>
>> * kmem limit set at A
>> * A and B empty taskwise
>> * bash in C does find /
>>
>> Because kmem_accounted is a boolean that was not set for C, no accounting

>> would be done. This is, however, not what we expect.
>>
>
> Hmm....do we need this new routines even while we have mem_cgroup_iter() ?
>
> Doesn't this work ?
>
> struct mem_cgroup {
>   .....
>   bool kmem_accounted_this;
>   atomic_t kmem_accounted;
>   ....
> }
>
> at set limit
>
> ....set_limit(memcg) {
>
>   if (newly accounted) {
>   mem_cgroup_iter() {
>   atomic_inc(&iter->kmem_accounted)
>   }
>   } else {
>   mem_cgroup_iter() {
>   atomic_dec(&iter->kmem_accounted);
>   }
> }
>
>
> hm ? Then, you can see kmem is accounted or not by
atomic_read(&memcg->kmem_accounted);
>

Accounted by itself / parent is still useful, and I see no reason to use
an atomic + bool if we can use a pair of bits.

As for the routine, I guess mem_cgroup_iter will work... It does a lot
more than I need, but for the sake of using what's already in there, I
can switch to it with no problems.

Subject: Re: [PATCH v4 05/25] memcg: Always free struct memcg through schedule_work()
Posted by KAMEZAWA Hiroyuki on Tue, 19 Jun 2012 00:11:20 GMT
View Forum Message <> Reply to Message

(2012/06/18 21:10), Glauber Costa wrote:
> On 06/18/2012 04:07 PM, Kamezawa Hiroyuki wrote:

>> (2012/06/18 19:27), Glauber Costa wrote:
>>> Right now we free struct memcg with kfree right after a
>>> rcu grace period, but defer it if we need to use vfree() to get
>>> rid of that memory area. We do that by need, because we need vfree
>>> to be called in a process context.
>>>
>>> This patch unifies this behavior, by ensuring that even kfree will
>>> happen in a separate thread. The goal is to have a stable place to
>>> call the upcoming jump label destruction function outside the realm
>>> of the complicated and quite far-reaching cgroup lock (that can't be
>>> held when calling neither the cpu_hotplug.lock nor the jump_label_mutex)
>>>
>>> Signed-off-by: Glauber Costa<glommer@parallels.com>
>>> CC: Tejun Heo<tj@kernel.org>
>>> CC: Li Zefan<lizefan@huawei.com>
>>> CC: Kamezawa Hiroyuki<kamezawa.hiroyu@jp.fujitsu.com>
>>> CC: Johannes Weiner<hannes@cmpxchg.org>
>>> CC: Michal Hocko<mhocko@suse.cz>
>>
>> How about cut out this patch and merge first as simple cleanu up and
>> to reduce patch stack on your side ?
>>
>> Acked-by: KAMEZAWA Hiroyuki<kamezawa.hiroyu@jp.fujitsu.com>
>
> I believe this is already in the -mm tree (from the sock memcg fixes)
>
> But actually, my main trouble with this series here, is that I am basing
> it on Pekka's tree, while some of the fixes are in -mm already.
> If I'd base it on -mm I would lose some of the stuff as well.
>
> Maybe Pekka can merge the current -mm with his tree?
>
> So far I am happy with getting comments from people about the code, so I
> did not get overly concerned about that.
>

Sure. thank you.
-Kame

Subject: Re: [PATCH v4 23/25] memcg: propagate kmem limiting information to children
Posted by KAMEZAWA Hiroyuki on Tue, 19 Jun 2012 00:16:20 GMT
View Forum Message <> Reply to Message

(2012/06/18 21:43), Glauber Costa wrote:
> On 06/18/2012 04:37 PM, Kamezawa Hiroyuki wrote:
>> (2012/06/18 19:28), Glauber Costa wrote:

>>> The current memcg slab cache management fails to present satisfatory hierarchical
>>> behavior in the following scenario:
>>>
>>> ->   /cgroups/memory/A/B/C
>>>
>>> * kmem limit set at A
>>> * A and B empty taskwise
>>> * bash in C does find /
>>>
>>> Because kmem_accounted is a boolean that was not set for C, no accounting
>>> would be done. This is, however, not what we expect.
>>>
>>
>> Hmm....do we need this new routines even while we have mem_cgroup_iter() ?
>>
>> Doesn't this work ?
>>
>> struct mem_cgroup {
>>   .....
>>   bool kmem_accounted_this;
>>   atomic_t kmem_accounted;
>>   ....
>> }
>>
>> at set limit
>>
>> ....set_limit(memcg) {
>>
>>   if (newly accounted) {
>>   mem_cgroup_iter() {
>>   atomic_inc(&iter->kmem_accounted)
>>   }
>>   } else {
>>   mem_cgroup_iter() {
>>   atomic_dec(&iter->kmem_accounted);
>>   }
>> }
>>
>>
>> hm ? Then, you can see kmem is accounted or not by
atomic_read(&memcg->kmem_accounted);
>>
>
> Accounted by itself / parent is still useful, and I see no reason to use
> an atomic + bool if we can use a pair of bits.
>
> As for the routine, I guess mem_cgroup_iter will work... It does a lot
> more than I need, but for the sake of using what's already in there, I

> can switch to it with no problems.
>

Hmm. please start from reusing existing routines.
If it's not enough, some enhancement for generic cgroup will be welcomed
rather than completely new one only for memcg.

Thanks,
-Kame

Subject: Re: [PATCH v4 23/25] memcg: propagate kmem limiting information to children
Posted by Glauber Costa on Tue, 19 Jun 2012 08:35:05 GMT
View Forum Message <> Reply to Message

On 06/19/2012 04:16 AM, Kamezawa Hiroyuki wrote:
> (2012/06/18 21:43), Glauber Costa wrote:
>> On 06/18/2012 04:37 PM, Kamezawa Hiroyuki wrote:
>>> (2012/06/18 19:28), Glauber Costa wrote:
>>>> The current memcg slab cache management fails to present satisfatory hierarchical
>>>> behavior in the following scenario:
>>>>
>>>> ->   /cgroups/memory/A/B/C
>>>>
>>>> * kmem limit set at A
>>>> * A and B empty taskwise
>>>> * bash in C does find /
>>>>
>>>> Because kmem_accounted is a boolean that was not set for C, no accounting
>>>> would be done. This is, however, not what we expect.
>>>>
>>>
>>> Hmm....do we need this new routines even while we have mem_cgroup_iter() ?
>>>
>>> Doesn't this work ?
>>>
>>> struct mem_cgroup {
>>>   .....
>>>   bool kmem_accounted_this;
>>>   atomic_t kmem_accounted;
>>>   ....
>>> }
>>>
>>> at set limit
>>>
>>> ....set_limit(memcg) {
>>>

>>> if (newly accounted) {
>>>   mem_cgroup_iter() {
>>>   atomic_inc(&iter->kmem_accounted)
>>>   }
>>> } else {
>>>   mem_cgroup_iter() {
>>>   atomic_dec(&iter->kmem_accounted);
>>>   }
>>> }
>>>
>>>
>>> hm ? Then, you can see kmem is accounted or not by
atomic_read(&memcg->kmem_accounted);
>>>
>>
>> Accounted by itself / parent is still useful, and I see no reason to use
>> an atomic + bool if we can use a pair of bits.
>>
>> As for the routine, I guess mem_cgroup_iter will work... It does a lot
>> more than I need, but for the sake of using what's already in there, I
>> can switch to it with no problems.
>>
>
> Hmm. please start from reusing existing routines.
> If it's not enough, some enhancement for generic cgroup will be welcomed
> rather than completely new one only for memcg.
>

And now that I am trying to adapt the code to the new function, I
remember clearly why I done this way. Sorry for my failed memory.

That has to do with the order of the walk. I need to enforce hierarchy,
which means whenever a cgroup has !use_hierarchy, I need to cut out that
branch, but continue scanning the tree for other branches.

That is a lot easier to do with depth-search tree walks like the one
proposed in this patch. for_each_mem_cgroup() seems to walk the tree in
css-creation order. Which means we need to keep track of parents that
has hierarchy disabled at all times (can be many), and always test for
ancestorship - which is expensive, but I don't particularly care.

But I'll give another shot with this one.

Subject: Re: [PATCH v4 23/25] memcg: propagate kmem limiting information to children
Posted by Glauber Costa on Tue, 19 Jun 2012 08:54:35 GMT

On 06/19/2012 12:35 PM, Glauber Costa wrote:
> On 06/19/2012 04:16 AM, Kamezawa Hiroyuki wrote:
>> (2012/06/18 21:43), Glauber Costa wrote:
>>> On 06/18/2012 04:37 PM, Kamezawa Hiroyuki wrote:
>>>> (2012/06/18 19:28), Glauber Costa wrote:
>>>>> The current memcg slab cache management fails to present satisfatory hierarchical
>>>>> behavior in the following scenario:
>>>>>
>>>>> ->   /cgroups/memory/A/B/C
>>>>>
>>>>> * kmem limit set at A
>>>>> * A and B empty taskwise
>>>>> * bash in C does find /
>>>>>
>>>>> Because kmem_accounted is a boolean that was not set for C, no accounting
>>>>> would be done. This is, however, not what we expect.
>>>>>
>>>>
>>>> Hmm....do we need this new routines even while we have mem_cgroup_iter() ?
>>>>
>>>> Doesn't this work ?
>>>>
>>>> struct mem_cgroup {
>>>>   .....
>>>>   bool kmem_accounted_this;
>>>>   atomic_t kmem_accounted;
>>>>   ....
>>>> }
>>>>
>>>> at set limit
>>>>
>>>> ....set_limit(memcg) {
>>>>
>>>>   if (newly accounted) {
>>>>   mem_cgroup_iter() {
>>>>   atomic_inc(&iter->kmem_accounted)
>>>>   }
>>>>   } else {
>>>>   mem_cgroup_iter() {
>>>>   atomic_dec(&iter->kmem_accounted);
>>>>   }
>>>> }
>>>>
>>>>
>>>> hm ? Then, you can see kmem is accounted or not by
atomic_read(&memcg->kmem_accounted);
>>>>

>>>
>>> Accounted by itself / parent is still useful, and I see no reason to use
>>> an atomic + bool if we can use a pair of bits.
>>>
>>> As for the routine, I guess mem_cgroup_iter will work... It does a lot
>>> more than I need, but for the sake of using what's already in there, I
>>> can switch to it with no problems.
>>>
>>
>> Hmm. please start from reusing existing routines.
>> If it's not enough, some enhancement for generic cgroup will be welcomed
>> rather than completely new one only for memcg.
>>
>
> And now that I am trying to adapt the code to the new function, I
> remember clearly why I done this way. Sorry for my failed memory.
>
> That has to do with the order of the walk. I need to enforce hierarchy,
> which means whenever a cgroup has !use_hierarchy, I need to cut out that
> branch, but continue scanning the tree for other branches.
>
> That is a lot easier to do with depth-search tree walks like the one
> proposed in this patch. for_each_mem_cgroup() seems to walk the tree in
> css-creation order. Which means we need to keep track of parents that
> has hierarchy disabled at all times (can be many), and always test for
> ancestorship - which is expensive, but I don't particularly care.
>
> But I'll give another shot with this one.
>

Humm, silly me. I was believing the hierarchical settings to be more
flexible than they really are.

I thought that it could be possible for a children of a parent with
use_hierarchy = 1 to have use_hierarchy = 0.

It seems not to be the case. This makes my life a lot easier.

Subject: Re: [PATCH v4 05/25] memcg: Always free struct memcg through schedule_work()
Posted by Pekka Enberg on Wed, 20 Jun 2012 07:32:50 GMT
View Forum Message <> Reply to Message

On Mon, 18 Jun 2012, Glauber Costa wrote:
> I believe this is already in the -mm tree (from the sock memcg fixes)
>
> But actually, my main trouble with this series here, is that I am basing

> it on Pekka's tree, while some of the fixes are in -mm already.
> If I'd base it on -mm I would lose some of the stuff as well.
>
> Maybe Pekka can merge the current -mm with his tree?

I first want to have a stable base from Christoph's "common slab" series
before I am comfortable with going forward with the memcg parts.

Feel free to push forward any preparational patches to the slab
allocators, though.

Pekka

Subject: Re: [PATCH v4 05/25] memcg: Always free struct memcg through schedule_work()
Posted by Glauber Costa on Wed, 20 Jun 2012 08:40:11 GMT
View Forum Message <> Reply to Message

On 06/20/2012 11:32 AM, Pekka Enberg wrote:
>> >Maybe Pekka can merge the current -mm with his tree?
> I first want to have a stable base from Christoph's "common slab" series
> before I am comfortable with going forward with the memcg parts.
>
> Feel free to push forward any preparational patches to the slab
> allocators, though.
>
> Pekka

Kame and others:

If you are already comfortable with the general shape of the series, it
would do me good to do the same with the memcg preparation patches, so
we have less code to review and merge in the next window.

They are:

memcg: Make it possible to use the stock for more than one page.
memcg: Reclaim when more than one page needed.
memcg: change defines to an enum

Do you see any value in merging them now ?

Subject: Re: [PATCH v4 23/25] memcg: propagate kmem limiting information to children
Posted by Glauber Costa on Wed, 20 Jun 2012 08:59:46 GMT

On 06/19/2012 12:54 PM, Glauber Costa wrote:
> On 06/19/2012 12:35 PM, Glauber Costa wrote:
>> On 06/19/2012 04:16 AM, Kamezawa Hiroyuki wrote:
>>> (2012/06/18 21:43), Glauber Costa wrote:
>>>> On 06/18/2012 04:37 PM, Kamezawa Hiroyuki wrote:
>>>>> (2012/06/18 19:28), Glauber Costa wrote:
>>>>>> The current memcg slab cache management fails to present satisfatory hierarchical
>>>>>> behavior in the following scenario:
>>>>>>
>>>>>> ->   /cgroups/memory/A/B/C
>>>>>>
>>>>>> * kmem limit set at A
>>>>>> * A and B empty taskwise
>>>>>> * bash in C does find /
>>>>>>
>>>>>> Because kmem_accounted is a boolean that was not set for C, no accounting
>>>>>> would be done. This is, however, not what we expect.
>>>>>>
>>>>>
>>>>> Hmm....do we need this new routines even while we have mem_cgroup_iter() ?
>>>>>
>>>>> Doesn't this work ?
>>>>>
>>>>> struct mem_cgroup {
>>>>>   .....
>>>>>   bool kmem_accounted_this;
>>>>>   atomic_t kmem_accounted;
>>>>>   ....
>>>>> }
>>>>>
>>>>> at set limit
>>>>>
>>>>> ....set_limit(memcg) {
>>>>>
>>>>>   if (newly accounted) {
>>>>>   mem_cgroup_iter() {
>>>>>   atomic_inc(&iter->kmem_accounted)
>>>>>   }
>>>>>   } else {
>>>>>   mem_cgroup_iter() {
>>>>>   atomic_dec(&iter->kmem_accounted);
>>>>>   }
>>>>> }
>>>>>
>>>>>
>>>>> hm ? Then, you can see kmem is accounted or not by
atomic_read(&memcg->kmem_accounted);

>>>>>
>>>>
>>>> Accounted by itself / parent is still useful, and I see no reason to use
>>>> an atomic + bool if we can use a pair of bits.
>>>>
>>>> As for the routine, I guess mem_cgroup_iter will work... It does a lot
>>>> more than I need, but for the sake of using what's already in there, I
>>>> can switch to it with no problems.
>>>>
>>>
>>> Hmm. please start from reusing existing routines.
>>> If it's not enough, some enhancement for generic cgroup will be welcomed
>>> rather than completely new one only for memcg.
>>>
>>
>> And now that I am trying to adapt the code to the new function, I
>> remember clearly why I done this way. Sorry for my failed memory.
>>
>> That has to do with the order of the walk. I need to enforce hierarchy,
>> which means whenever a cgroup has !use_hierarchy, I need to cut out that
>> branch, but continue scanning the tree for other branches.
>>
>> That is a lot easier to do with depth-search tree walks like the one
>> proposed in this patch. for_each_mem_cgroup() seems to walk the tree in
>> css-creation order. Which means we need to keep track of parents that
>> has hierarchy disabled at all times (can be many), and always test for
>> ancestorship - which is expensive, but I don't particularly care.
>>
>> But I'll give another shot with this one.
>>
>
> Humm, silly me. I was believing the hierarchical settings to be more
> flexible than they really are.
>
> I thought that it could be possible for a children of a parent with
> use_hierarchy = 1 to have use_hierarchy = 0.
>
> It seems not to be the case. This makes my life a lot easier.
>

How about the following patch?

It is still expensive in the clear_bit case, because I can't just walk
the whole tree flipping the bit down: I need to stop whenever I see a
branch whose root is itself accounted - and the ordering of iter forces
me to always check the tree up (So we got O(n*h) h being height instead
of O(n)).

for flipping the bit up, it is easy enough.

File Attachments
1)
0001-memcg-propagate-kmem-limiting-information-to-childre.patch, downloaded 838 times

Subject: Re: [PATCH v4 08/25] memcg: change defines to an enum
Posted by Michal Hocko on Wed, 20 Jun 2012 13:13:22 GMT
View Forum Message <> Reply to Message

[Sorry for late reply. I am aware of the series, I am just too busy to
give it serious time needed for review. It doesn't make much sense to
delay these preparational pieces so...]

On Mon 18-06-12 14:28:01, 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.
>
> Signed-off-by: Glauber Costa <glommer@parallels.com>
> CC: Michal Hocko <mhocko@suse.cz>
> CC: Johannes Weiner <hannes@cmpxchg.org>
> Acked-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>

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

> ---
> mm/memcontrol.c | 9 ++++++---
> 1 file changed, 6 insertions(+), 3 deletions(-)
>
> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
> index b6cb075..cc1fdb4 100644
> --- a/mm/memcontrol.c
> +++ b/mm/memcontrol.c
> @@ -374,9 +374,12 @@ 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)
> --
> 1.7.10.2
>
> --
> To unsubscribe, send a message with 'unsubscribe linux-mm' in
> the body to majordomo@kvack.org. For more info on Linux MM,
> see: http://www.linux-mm.org/ .
> Don't email: email@kvack.org

--
Michal Hocko
SUSE Labs
SUSE LINUX s.r.o.
Lihovarska 1060/12
190 00 Praha 9
Czech Republic

Subject: Re: [PATCH v4 05/25] memcg: Always free struct memcg through schedule_work()
Posted by Michal Hocko on Wed, 20 Jun 2012 13:20:50 GMT
View Forum Message <> Reply to Message

On Mon 18-06-12 14:27:58, Glauber Costa wrote:
> Right now we free struct memcg with kfree right after a
> rcu grace period, but defer it if we need to use vfree() to get
> rid of that memory area. We do that by need, because we need vfree
> to be called in a process context.
>
> This patch unifies this behavior, by ensuring that even kfree will
> happen in a separate thread. The goal is to have a stable place to
> call the upcoming jump label destruction function outside the realm
> of the complicated and quite far-reaching cgroup lock (that can't be
> held when calling neither the cpu_hotplug.lock nor the jump_label_mutex)

This one is in memcg-devel (mmotm) tree for quite some time with acks
from me and Kamezawa.

> Signed-off-by: Glauber Costa <glommer@parallels.com>
> CC: Tejun Heo <tj@kernel.org>
> CC: Li Zefan <lizefan@huawei.com>
> CC: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
> CC: Johannes Weiner <hannes@cmpxchg.org>
> CC: Michal Hocko <mhocko@suse.cz>

> ---
> mm/memcontrol.c | 24 ++++++++++++++++++-----------
> 1 file changed, 13 insertions(+), 11 deletions(-)
>
> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
> index e3b528e..ce15be4 100644
> --- a/mm/memcontrol.c
> +++ b/mm/memcontrol.c
> @@ -245,8 +245,8 @@ struct mem_cgroup {
>   */
>   struct rcu_head rcu_freeing;
>   /*
> -   * But when using vfree(), that cannot be done at
> -   * interrupt time, so we must then queue the work.
> +   * We also need some space for a worker in deferred freeing.
> +   * By the time we call it, rcu_freeing is not longer in use.
>   */
>   struct work_struct work_freeing;
> };
> @@ -4826,23 +4826,28 @@ out_free:
> }
>
> /*
> - * Helpers for freeing a vzalloc()ed mem_cgroup by RCU,
> + * 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 vfree_work(struct work_struct *work)
> +static void free_work(struct work_struct *work)
> {
>   struct mem_cgroup *memcg;
> + int size = sizeof(struct mem_cgroup);
>
>   memcg = container_of(work, struct mem_cgroup, work_freeing);
> - vfree(memcg);
> + if (size < PAGE_SIZE)
> + kfree(memcg);
> + else
> + vfree(memcg);
> }
> -static void vfree_rcu(struct rcu_head *rcu_head)
> +
> +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, vfree_work);
> + INIT_WORK(&memcg->work_freeing, free_work);
> schedule_work(&memcg->work_freeing);
> }
>
> @@ -4868,10 +4873,7 @@ static void __mem_cgroup_free(struct mem_cgroup *memcg)
> free_mem_cgroup_per_zone_info(memcg, node);
>
> free_percpu(memcg->stat);
> - if (sizeof(struct mem_cgroup) < PAGE_SIZE)
> - kfree_rcu(memcg, rcu_freeing);
> - else
> - call_rcu(&memcg->rcu_freeing, vfree_rcu);
> + call_rcu(&memcg->rcu_freeing, free_rcu);
> }
>
> static void mem_cgroup_get(struct mem_cgroup *memcg)
> --
> 1.7.10.2
>
> --
> To unsubscribe from this list: send the line "unsubscribe linux-kernel" in
> the body of a message to majordomo@vger.kernel.org
> More majordomo info at http://vger.kernel.org/majordomo-info.html
> Please read the FAQ at http://www.tux.org/lkml/

--
Michal Hocko
SUSE Labs
SUSE LINUX s.r.o.
Lihovarska 1060/12
190 00 Praha 9
Czech Republic

Subject: Re: [PATCH v4 06/25] memcg: Make it possible to use the stock for more than one page.
Posted by Michal Hocko on Wed, 20 Jun 2012 13:28:04 GMT
View Forum Message <> Reply to Message

On Mon 18-06-12 14:27:59, Glauber Costa wrote:
> From: Suleiman Souhlal <ssouhlal@FreeBSD.org>
>
> Signed-off-by: Suleiman Souhlal <suleiman@google.com>
> Signed-off-by: Glauber Costa <glommer@parallels.com>
> Acked-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>

I am not sure the patch is good to merge on its own without the rest.

One comment bellow.

> ---
> mm/memcontrol.c | 18 +++++++++---------
> 1 file changed, 9 insertions(+), 9 deletions(-)
>
> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
> index ce15be4..00b9f1e 100644
> --- a/mm/memcontrol.c
> +++ b/mm/memcontrol.c
> @@ -1998,19 +1998,19 @@ 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.
> + * Try to consume stocked charge on this cpu. If success, nr_pages pages are
> + * 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.
>   */
> -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;

I guess you want:
 if (nr_pages > CHARGE_BATCH)
  return false;

because you don't want to try to use stock for THP pages.

>
> 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);
> @@ -2309,7 +2309,7 @@ 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 {
> @@ -2334,7 +2334,7 @@ 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
> --
> 1.7.10.2
>
> --
> To unsubscribe, send a message with 'unsubscribe linux-mm' in
> the body to majordomo@kvack.org. For more info on Linux MM,
> see: http://www.linux-mm.org/ .
> Don't email: email@kvack.org

--
Michal Hocko
SUSE Labs
SUSE LINUX s.r.o.
Lihovarska 1060/12
190 00 Praha 9
Czech Republic

Subject: Re: [PATCH v4 07/25] memcg: Reclaim when more than one page needed.
Posted by Michal Hocko on Wed, 20 Jun 2012 13:47:38 GMT
View Forum Message <> Reply to Message

On Mon 18-06-12 14:28:00, Glauber Costa wrote:
> From: Suleiman Souhlal <ssouhlal@FreeBSD.org>
>
> mem_cgroup_do_charge() was written before slab accounting, and expects
> three cases: being called for 1 page, being called for a stock of 32 pages,
> or being called for a hugepage. If we call for 2 or 3 pages (and several
> slabs used in process creation are such, at least with the debug options I
> had), it assumed it's being called for stock and just retried without reclaiming.
>
> Fix that by passing down a minsize argument in addition to the csize.

>
> And what to do about that (csize == PAGE_SIZE && ret) retry? If it's
> needed at all (and presumably is since it's there, perhaps to handle
> races), then it should be extended to more than PAGE_SIZE, yet how far?
> And should there be a retry count limit, of what? For now retry up to
> COSTLY_ORDER (as page_alloc.c does), stay safe with a cond_resched(),
> and make sure not to do it if __GFP_NORETRY.
>
> [v4: fixed nr pages calculation pointed out by Christoph Lameter]
>
> Signed-off-by: Suleiman Souhlal <suleiman@google.com>
> Signed-off-by: Glauber Costa <glommer@parallels.com>
> Reviewed-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>

I think this is not ready to be merged yet.
Two comments below.

[...]
> @@ -2210,18 +2211,18 @@ static int mem_cgroup_do_charge(struct mem_cgroup *memcg,
gfp_t gfp_mask,
>   } else
>   mem_over_limit = mem_cgroup_from_res_counter(fail_res, res);
>   /*
> - * nr_pages can be either a huge page (HPAGE_PMD_NR), a batch
> - * of regular pages (CHARGE_BATCH), or a single regular page (1).
> - *
>   * Never reclaim on behalf of optional batching, retry with a
>   * single page instead.
>   */
> - if (nr_pages == CHARGE_BATCH)
> + if (nr_pages > min_pages)
>   return CHARGE_RETRY;
>
>   if (!(gfp_mask & __GFP_WAIT))
>   return CHARGE_WOULDBLOCK;
>
> + if (gfp_mask & __GFP_NORETRY)
> + return CHARGE_NOMEM;

This is kmem specific and should be preparated out in case this should
be merged before the rest.
Btw. I assume that oom==false when called from kmem...

> +
>   ret = mem_cgroup_reclaim(mem_over_limit, gfp_mask, flags);
>   if (mem_cgroup_margin(mem_over_limit) >= nr_pages)
>   return CHARGE_RETRY;
> @@ -2234,8 +2235,10 @@ static int mem_cgroup_do_charge(struct mem_cgroup *memcg,

gfp_t gfp_mask,
> * unlikely to succeed so close to the limit, and we fall back
> * to regular pages anyway in case of failure.
> */
> - if (nr_pages == 1 && ret)
> + if (nr_pages <= (1 << PAGE_ALLOC_COSTLY_ORDER) && ret) {
> + cond_resched();
> return CHARGE_RETRY;
> + }

What prevents us from looping for unbounded amount of time here?
Maybe you need to consider the number of reclaimed pages here.

>
> /*
> * At task move, charge accounts can be doubly counted. So, it's
> @@ -2369,7 +2372,8 @@ again:
> nr_oom_retries = MEM_CGROUP_RECLAIM_RETRIES;
> }
>
> - ret = mem_cgroup_do_charge(memcg, gfp_mask, batch, oom_check);
> + ret = mem_cgroup_do_charge(memcg, gfp_mask, batch, nr_pages,
> + oom_check);
> switch (ret) {
> case CHARGE_OK:
> break;
> --
> 1.7.10.2
>
> --
> To unsubscribe, send a message with 'unsubscribe linux-mm' in
> the body to majordomo@kvack.org. For more info on Linux MM,
> see: http://www.linux-mm.org/ .
> Don't email: email@kvack.org

--
Michal Hocko
SUSE Labs
SUSE LINUX s.r.o.
Lihovarska 1060/12
190 00 Praha 9
Czech Republic

Subject: Re: [PATCH v4 06/25] memcg: Make it possible to use the stock for more than one page.
Posted by Glauber Costa on Wed, 20 Jun 2012 19:36:47 GMT

On 06/20/2012 05:28 PM, Michal Hocko wrote:
> On Mon 18-06-12 14:27:59, Glauber Costa wrote:
>> From: Suleiman Souhlal <ssouhlal@FreeBSD.org>
>>
>> Signed-off-by: Suleiman Souhlal <suleiman@google.com>
>> Signed-off-by: Glauber Costa <glommer@parallels.com>
>> Acked-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
>
> I am not sure the patch is good to merge on its own without the rest.
> One comment bellow.
>
>> ---
>> mm/memcontrol.c | 18 +++++++++---------
>> 1 file changed, 9 insertions(+), 9 deletions(-)
>>
>> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
>> index ce15be4..00b9f1e 100644
>> --- a/mm/memcontrol.c
>> +++ b/mm/memcontrol.c
>> @@ -1998,19 +1998,19 @@ 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.
>> + * Try to consume stocked charge on this cpu. If success, nr_pages pages are
>> + * 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.
>>   */
>> -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;
>
> I guess you want:
> if (nr_pages > CHARGE_BATCH)
>   return false;
>
> because you don't want to try to use stock for THP pages.

The code reads:

+ if (memcg == stock->cached && stock->nr_pages >= nr_pages)
+ stock->nr_pages -= nr_pages;

Isn't stock->nr_pages always <= CHARGE_BATCH by definition?

Subject: Re: [PATCH v4 07/25] memcg: Reclaim when more than one page needed.
Posted by Glauber Costa on Wed, 20 Jun 2012 19:43:52 GMT
View Forum Message <> Reply to Message

On 06/20/2012 05:47 PM, Michal Hocko wrote:
> On Mon 18-06-12 14:28:00, Glauber Costa wrote:
>> From: Suleiman Souhlal <ssouhlal@FreeBSD.org>
>>
>> mem_cgroup_do_charge() was written before slab accounting, and expects
>> three cases: being called for 1 page, being called for a stock of 32 pages,
>> or being called for a hugepage. If we call for 2 or 3 pages (and several
>> slabs used in process creation are such, at least with the debug options I
>> had), it assumed it's being called for stock and just retried without reclaiming.
>>
>> Fix that by passing down a minsize argument in addition to the csize.
>>
>> And what to do about that (csize == PAGE_SIZE && ret) retry? If it's
>> needed at all (and presumably is since it's there, perhaps to handle
>> races), then it should be extended to more than PAGE_SIZE, yet how far?
>> And should there be a retry count limit, of what? For now retry up to
>> COSTLY_ORDER (as page_alloc.c does), stay safe with a cond_resched(),
>> and make sure not to do it if __GFP_NORETRY.
>>
>> [v4: fixed nr pages calculation pointed out by Christoph Lameter]
>>
>> Signed-off-by: Suleiman Souhlal <suleiman@google.com>
>> Signed-off-by: Glauber Costa <glommer@parallels.com>
>> Reviewed-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
>
> I think this is not ready to be merged yet.
Fair Enough

> Two comments below.
>
> [...]
>> @@ -2210,18 +2211,18 @@ static int mem_cgroup_do_charge(struct mem_cgroup *memcg, gfp_t gfp_mask,
>> } else
>> mem_over_limit = mem_cgroup_from_res_counter(fail_res, res);
>> /*

>> - * nr_pages can be either a huge page (HPAGE_PMD_NR), a batch
>> - * of regular pages (CHARGE_BATCH), or a single regular page (1).
>> - *
>>   * Never reclaim on behalf of optional batching, retry with a
>>   * single page instead.
>>   */
>> - if (nr_pages == CHARGE_BATCH)
>> + if (nr_pages > min_pages)
>>   return CHARGE_RETRY;
>>
>>   if (!(gfp_mask & __GFP_WAIT))
>>   return CHARGE_WOULDBLOCK;
>>
>> + if (gfp_mask & __GFP_NORETRY)
>> + return CHARGE_NOMEM;
>
> This is kmem specific and should be preparated out in case this should
> be merged before the rest.
ok.

> Btw. I assume that oom==false when called from kmem...

What prevents the oom killer to be called for a reclaimable kmem
allocation that can be satisfied ?

>> +
>>   ret = mem_cgroup_reclaim(mem_over_limit, gfp_mask, flags);
>>   if (mem_cgroup_margin(mem_over_limit) >= nr_pages)
>>   return CHARGE_RETRY;
>> @@ -2234,8 +2235,10 @@ static int mem_cgroup_do_charge(struct mem_cgroup *memcg,
gfp_t gfp_mask,
>>   * unlikely to succeed so close to the limit, and we fall back
>>   * to regular pages anyway in case of failure.
>>   */
>> - if (nr_pages == 1 && ret)
>> + if (nr_pages <= (1 << PAGE_ALLOC_COSTLY_ORDER) && ret) {
>> + cond_resched();
>>   return CHARGE_RETRY;
>> + }
>
> What prevents us from looping for unbounded amount of time here?
> Maybe you need to consider the number of reclaimed pages here.

Why would we even loop here? It will just return CHARGE_RETRY, it is up
to the caller to decide whether or not it will retry.

Subject: Re: [PATCH v4 05/25] memcg: Always free struct memcg through schedule_work()
Posted by KAMEZAWA Hiroyuki on Thu, 21 Jun 2012 11:39:14 GMT
View Forum Message <> Reply to Message

(2012/06/20 17:40), Glauber Costa wrote:
> On 06/20/2012 11:32 AM, Pekka Enberg wrote:
>>> >Maybe Pekka can merge the current -mm with his tree?
>> I first want to have a stable base from Christoph's "common slab" series
>> before I am comfortable with going forward with the memcg parts.
>>
>> Feel free to push forward any preparational patches to the slab
>> allocators, though.
>>
>> Pekka
>
> Kame and others:
>
> If you are already comfortable with the general shape of the series, it would do me good to do the same with the memcg preparation patches, so we have less code to review and merge in the next window.
>
> They are:
>
> memcg: Make it possible to use the stock for more than one page.
> memcg: Reclaim when more than one page needed.
> memcg: change defines to an enum
>
> Do you see any value in merging them now ?
>

I'll be okay with the 3 patches for memcg.

Thanks,
-Kame

Subject: Re: [PATCH v4 06/25] memcg: Make it possible to use the stock for more than one page.
Posted by Michal Hocko on Thu, 21 Jun 2012 21:14:52 GMT
View Forum Message <> Reply to Message

On Wed 20-06-12 23:36:47, Glauber Costa wrote:
> On 06/20/2012 05:28 PM, Michal Hocko wrote:
> >On Mon 18-06-12 14:27:59, Glauber Costa wrote:
> >>From: Suleiman Souhlal <ssouhlal@FreeBSD.org>
> >>
> >>Signed-off-by: Suleiman Souhlal <suleiman@google.com>

> >>Signed-off-by: Glauber Costa <glommer@parallels.com>
> >>Acked-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
> >
> >I am not sure the patch is good to merge on its own without the rest.
> >One comment bellow.
> >
> >>---
> >> mm/memcontrol.c | 18 +++++++++---------
> >> 1 file changed, 9 insertions(+), 9 deletions(-)
> >>
> >>diff --git a/mm/memcontrol.c b/mm/memcontrol.c
> >>index ce15be4..00b9f1e 100644
> >>--- a/mm/memcontrol.c
> >>+++ b/mm/memcontrol.c
> >>@@ -1998,19 +1998,19 @@ 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.
> >>+ * Try to consume stocked charge on this cpu. If success, nr_pages pages are
> >>+ * 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.
> >>   */
> >>-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;
> >
> >I guess you want:
> > if (nr_pages > CHARGE_BATCH)
> > return false;
> >
> >because you don't want to try to use stock for THP pages.
>
>
> The code reads:
>
> +   if (memcg == stock->cached && stock->nr_pages >= nr_pages)
> +   stock->nr_pages -= nr_pages;
>
> Isn't stock->nr_pages always <= CHARGE_BATCH by definition?

Yes it is, but why to disable preemption if we know this has no chance
to succeed at all?

--
Michal Hocko
SUSE Labs
SUSE LINUX s.r.o.
Lihovarska 1060/12
190 00 Praha 9
Czech Republic

Subject: Re: [PATCH v4 07/25] memcg: Reclaim when more than one page needed.
Posted by Michal Hocko on Thu, 21 Jun 2012 21:19:23 GMT
View Forum Message <> Reply to Message

On Wed 20-06-12 23:43:52, Glauber Costa wrote:
> On 06/20/2012 05:47 PM, Michal Hocko wrote:
> >On Mon 18-06-12 14:28:00, Glauber Costa wrote:
> >>From: Suleiman Souhlal <ssouhlal@FreeBSD.org>
> >>
> >>mem_cgroup_do_charge() was written before slab accounting, and expects
> >>three cases: being called for 1 page, being called for a stock of 32 pages,
> >>or being called for a hugepage. If we call for 2 or 3 pages (and several
> >>slabs used in process creation are such, at least with the debug options I
> >>had), it assumed it's being called for stock and just retried without reclaiming.
> >>
> >>Fix that by passing down a minsize argument in addition to the csize.
> >>
> >>And what to do about that (csize == PAGE_SIZE && ret) retry? If it's
> >>needed at all (and presumably is since it's there, perhaps to handle
> >>races), then it should be extended to more than PAGE_SIZE, yet how far?
> >>And should there be a retry count limit, of what? For now retry up to
> >>COSTLY_ORDER (as page_alloc.c does), stay safe with a cond_resched(),
> >>and make sure not to do it if __GFP_NORETRY.
> >>
> >>[v4: fixed nr pages calculation pointed out by Christoph Lameter]
> >>
> >>Signed-off-by: Suleiman Souhlal <suleiman@google.com>
> >>Signed-off-by: Glauber Costa <glommer@parallels.com>
> >>Reviewed-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
> >
> >I think this is not ready to be merged yet.
> Fair Enough
>
> >Two comments below.
> >

> >[...]
> >>@@ -2210,18 +2211,18 @@ static int mem_cgroup_do_charge(struct mem_cgroup
*memcg, gfp_t gfp_mask,
> >>   } else
> >>   mem_over_limit = mem_cgroup_from_res_counter(fail_res, res);
> >>   /*
> >>- * nr_pages can be either a huge page (HPAGE_PMD_NR), a batch
> >>- * of regular pages (CHARGE_BATCH), or a single regular page (1).
> >>- *
> >>   * Never reclaim on behalf of optional batching, retry with a
> >>   * single page instead.
> >>   */
> >>- if (nr_pages == CHARGE_BATCH)
> >>+ if (nr_pages > min_pages)
> >>   return CHARGE_RETRY;
> >>
> >>   if (!(gfp_mask & __GFP_WAIT))
> >>   return CHARGE_WOULDBLOCK;
> >>
> >>+ if (gfp_mask & __GFP_NORETRY)
> >>+ return CHARGE_NOMEM;
> >
> >This is kmem specific and should be preparated out in case this should
> >be merged before the rest.
> ok.
>
> >Btw. I assume that oom==false when called from kmem...
>
> What prevents the oom killer to be called for a reclaimable kmem
> allocation that can be satisfied ?

Well, I am not familiar with the rest of the patch series yet (sorry
about that) but playing with oom can be really nasty if oom score
doesn't consider also kmem allocations. You can end up killing
unexpected processes just because of kmem hungry (and nasty) process.
Dunno, have to thing about that.

> >>+
> >>   ret = mem_cgroup_reclaim(mem_over_limit, gfp_mask, flags);
> >>   if (mem_cgroup_margin(mem_over_limit) >= nr_pages)
> >>   return CHARGE_RETRY;
> >>@@ -2234,8 +2235,10 @@ static int mem_cgroup_do_charge(struct mem_cgroup *memcg,
gfp_t gfp_mask,
> >>   * unlikely to succeed so close to the limit, and we fall back
> >>   * to regular pages anyway in case of failure.
> >>   */
> >>- if (nr_pages == 1 && ret)
> >>+ if (nr_pages <= (1 << PAGE_ALLOC_COSTLY_ORDER) && ret) {

> >>+ cond_resched();
> >>   return CHARGE_RETRY;
> >>+ }
> >
> >What prevents us from looping for unbounded amount of time here?
> >Maybe you need to consider the number of reclaimed pages here.
>
> Why would we even loop here? It will just return CHARGE_RETRY, it is
> up to the caller to decide whether or not it will retry.

Yes, but the test was original to prevent oom when we managed to reclaim
something. And something might be enough for a single page but now you
have high order allocations so we can retry without any success.

--
Michal Hocko
SUSE Labs
SUSE LINUX s.r.o.
Lihovarska 1060/12
190 00 Praha 9
Czech Republic

Subject: Re: [PATCH v4 23/25] memcg: propagate kmem limiting information to children
Posted by KAMEZAWA Hiroyuki on Sat, 23 Jun 2012 04:19:37 GMT
View Forum Message <> Reply to Message

(2012/06/20 17:59), Glauber Costa wrote:
> On 06/19/2012 12:54 PM, Glauber Costa wrote:
>> On 06/19/2012 12:35 PM, Glauber Costa wrote:
>>> On 06/19/2012 04:16 AM, Kamezawa Hiroyuki wrote:
>>>> (2012/06/18 21:43), Glauber Costa wrote:
>>>>> On 06/18/2012 04:37 PM, Kamezawa Hiroyuki wrote:
>>>>>> (2012/06/18 19:28), Glauber Costa wrote:
>>>>>>> The current memcg slab cache management fails to present satisfatory hierarchical
>>>>>>> behavior in the following scenario:
>>>>>>>
>>>>>>> ->   /cgroups/memory/A/B/C
>>>>>>>
>>>>>>> * kmem limit set at A
>>>>>>> * A and B empty taskwise
>>>>>>> * bash in C does find /
>>>>>>>
>>>>>>> Because kmem_accounted is a boolean that was not set for C, no accounting
>>>>>>> would be done. This is, however, not what we expect.
>>>>>>>
>>>>>>

>>>>>> Hmm....do we need this new routines even while we have mem_cgroup_iter() ?
>>>>>>
>>>>>> Doesn't this work ?
>>>>>>
>>>>>> struct mem_cgroup {
>>>>>>   .....
>>>>>>   bool kmem_accounted_this;
>>>>>>   atomic_t kmem_accounted;
>>>>>>   ....
>>>>>> }
>>>>>>
>>>>>> at set limit
>>>>>>
>>>>>> ....set_limit(memcg) {
>>>>>>
>>>>>>   if (newly accounted) {
>>>>>>   mem_cgroup_iter() {
>>>>>>   atomic_inc(&iter->kmem_accounted)
>>>>>>   }
>>>>>>   } else {
>>>>>>   mem_cgroup_iter() {
>>>>>>   atomic_dec(&iter->kmem_accounted);
>>>>>>   }
>>>>>> }
>>>>>>
>>>>>>
>>>>>> hm ? Then, you can see kmem is accounted or not by
atomic_read(&memcg->kmem_accounted);
>>>>>>
>>>>>
>>>>> Accounted by itself / parent is still useful, and I see no reason to use
>>>>> an atomic + bool if we can use a pair of bits.
>>>>>
>>>>> As for the routine, I guess mem_cgroup_iter will work... It does a lot
>>>>> more than I need, but for the sake of using what's already in there, I
>>>>> can switch to it with no problems.
>>>>>
>>>>
>>>> Hmm. please start from reusing existing routines.
>>>> If it's not enough, some enhancement for generic cgroup will be welcomed
>>>> rather than completely new one only for memcg.
>>>>
>>>
>>> And now that I am trying to adapt the code to the new function, I
>>> remember clearly why I done this way. Sorry for my failed memory.
>>>
>>> That has to do with the order of the walk. I need to enforce hierarchy,
>>> which means whenever a cgroup has !use_hierarchy, I need to cut out that

>>> branch, but continue scanning the tree for other branches.
>>>
>>> That is a lot easier to do with depth-search tree walks like the one
>>> proposed in this patch. for_each_mem_cgroup() seems to walk the tree in
>>> css-creation order. Which means we need to keep track of parents that
>>> has hierarchy disabled at all times (can be many), and always test for
>>> ancestorship - which is expensive, but I don't particularly care.
>>>
>>> But I'll give another shot with this one.
>>>
>>
>> Humm, silly me. I was believing the hierarchical settings to be more
>> flexible than they really are.
>>
>> I thought that it could be possible for a children of a parent with
>> use_hierarchy = 1 to have use_hierarchy = 0.
>>
>> It seems not to be the case. This makes my life a lot easier.
>>
>
> How about the following patch?
>
> It is still expensive in the clear_bit case, because I can't just walk
> the whole tree flipping the bit down: I need to stop whenever I see a
> branch whose root is itself accounted - and the ordering of iter forces
> me to always check the tree up (So we got O(n*h) h being height instead
> of O(n)).
>
> for flipping the bit up, it is easy enough.
>
>
Yes. It seems much nicer.

Thanks,
-Kame

Subject: Re: [PATCH v4 06/25] memcg: Make it possible to use the stock for more than one page.
Posted by Glauber Costa on Mon, 25 Jun 2012 13:03:04 GMT
View Forum Message <> Reply to Message

On 06/20/2012 05:28 PM, Michal Hocko wrote:
> I guess you want:
> if (nr_pages > CHARGE_BATCH)
>   return false;
>
> because you don't want to try to use stock for THP pages.

Done, thanks.

Subject: Re: [PATCH v4 07/25] memcg: Reclaim when more than one page needed.
Posted by Glauber Costa on Mon, 25 Jun 2012 13:13:53 GMT
View Forum Message <> Reply to Message

>>>> +
>>>> ret = mem_cgroup_reclaim(mem_over_limit, gfp_mask, flags);
>>>> if (mem_cgroup_margin(mem_over_limit) >= nr_pages)
>>>> return CHARGE_RETRY;
>>>> @@ -2234,8 +2235,10 @@ static int mem_cgroup_do_charge(struct mem_cgroup
*memcg, gfp_t gfp_mask,
>>>> * unlikely to succeed so close to the limit, and we fall back
>>>> * to regular pages anyway in case of failure.
>>>> */
>>>> - if (nr_pages == 1 && ret)
>>>> + if (nr_pages <= (1 << PAGE_ALLOC_COSTLY_ORDER) && ret) {
>>>> + cond_resched();
>>>> return CHARGE_RETRY;
>>>> + }
>>>
>>> What prevents us from looping for unbounded amount of time here?
>>> Maybe you need to consider the number of reclaimed pages here.
>>
>> Why would we even loop here? It will just return CHARGE_RETRY, it is
>> up to the caller to decide whether or not it will retry.
>
> Yes, but the test was original to prevent oom when we managed to reclaim
> something. And something might be enough for a single page but now you
> have high order allocations so we can retry without any success.
>

So,

Most of the kmem allocations are likely to be quite small as well. For
the slab, we're dealing with the order of 2-3 pages, and for other
allocations that may happen, like stack, they will be in the order of 2
pages as well.

So one thing I could do here, is define a threshold, say, 3, and only
retry for that very low threshold, instead of following COSTLY_ORDER.
I don't expect two or three pages to be much less likely to be freed
than a single page.

I am fine with ripping of the cond_resched as well.

Let me know if you would be okay with that.

Subject: Re: [PATCH v4 07/25] memcg: Reclaim when more than one page needed.
Posted by Glauber Costa on Mon, 25 Jun 2012 14:04:08 GMT
View Forum Message <> Reply to Message

On 06/25/2012 05:13 PM, Glauber Costa wrote:
>
>>>>> +
>>>>> ret = mem_cgroup_reclaim(mem_over_limit, gfp_mask, flags);
>>>>> if (mem_cgroup_margin(mem_over_limit) >= nr_pages)
>>>>> return CHARGE_RETRY;
>>>>> @@ -2234,8 +2235,10 @@ static int mem_cgroup_do_charge(struct
>>>>> mem_cgroup *memcg, gfp_t gfp_mask,
>>>>> * unlikely to succeed so close to the limit, and we fall back
>>>>> * to regular pages anyway in case of failure.
>>>>> */
>>>>> - if (nr_pages == 1 && ret)
>>>>> + if (nr_pages <= (1 << PAGE_ALLOC_COSTLY_ORDER) && ret) {
>>>>> + cond_resched();
>>>>> return CHARGE_RETRY;
>>>>> + }
>>>>
>>>> What prevents us from looping for unbounded amount of time here?
>>>> Maybe you need to consider the number of reclaimed pages here.
>>>
>>> Why would we even loop here? It will just return CHARGE_RETRY, it is
>>> up to the caller to decide whether or not it will retry.
>>
>> Yes, but the test was original to prevent oom when we managed to reclaim
>> something. And something might be enough for a single page but now you
>> have high order allocations so we can retry without any success.
>>
>
> So,
>
> Most of the kmem allocations are likely to be quite small as well. For
> the slab, we're dealing with the order of 2-3 pages, and for other
> allocations that may happen, like stack, they will be in the order of 2
> pages as well.
>
> So one thing I could do here, is define a threshold, say, 3, and only
> retry for that very low threshold, instead of following COSTLY_ORDER.
> I don't expect two or three pages to be much less likely to be freed
> than a single page.

>
> I am fine with ripping of the cond_resched as well.
>
> Let me know if you would be okay with that.
>
>

For the record, here's the patch I would propose.

At this point, I think it would be nice to Suleiman to say if he is
still okay with the changes.

File Attachments

1) 0001-memcg-Reclaim-when-more-than-one-page-needed.patch, downloaded 753 times

---

Subject: Re: [PATCH v4 24/25] memcg/slub: shrink dead caches
Posted by Christoph Lameter on Fri, 06 Jul 2012 15:16:36 GMT
View Forum Message <> Reply to Message

On Mon, 18 Jun 2012, Glauber Costa wrote:

> In the slub allocator, when the last object of a page goes away, we
> don't necessarily free it - there is not necessarily a test for empty
> page in any slab_free path.

This is the same btw in SLAB which keeps objects in per cpu caches and
keeps empty slab pages on special queues.

> This patch marks all memcg caches as dead. kmem_cache_shrink is called
> for the ones who are not yet dead - this will force internal cache
> reorganization, and then all references to empty pages will be removed.

You need to call this also for slab to drain the caches and free the pages
on the empty list.

Subject: Re: [PATCH v4 24/25] memcg/slub: shrink dead caches
Posted by Glauber Costa on Fri, 20 Jul 2012 22:16:56 GMT
View Forum Message <> Reply to Message

On 07/06/2012 12:16 PM, Christoph Lameter wrote:
> On Mon, 18 Jun 2012, Glauber Costa wrote:
>
>> In the slub allocator, when the last object of a page goes away, we
>> don't necessarily free it - there is not necessarily a test for empty

>> page in any slab_free path.
>
> This is the same btw in SLAB which keeps objects in per cpu caches and
> keeps empty slab pages on special queues.
>
>> This patch marks all memcg caches as dead. kmem_cache_shrink is called
>> for the ones who are not yet dead - this will force internal cache
>> reorganization, and then all references to empty pages will be removed.
>
> You need to call this also for slab to drain the caches and free the pages
> on the empty list.
>
Doesn't the SLAB have a time-based reaper for that?

That's why I was less concerned with the SLAB, but I can certainly call
it for both.

Subject: Re: [PATCH v4 24/25] memcg/slub: shrink dead caches
Posted by Christoph Lameter on Wed, 25 Jul 2012 15:23:40 GMT
View Forum Message <> Reply to Message

On Fri, 20 Jul 2012, Glauber Costa wrote:

> > This is the same btw in SLAB which keeps objects in per cpu caches and
> > keeps empty slab pages on special queues.
> >
> >> This patch marks all memcg caches as dead. kmem_cache_shrink is called
> >> for the ones who are not yet dead - this will force internal cache
> >> reorganization, and then all references to empty pages will be removed.
> >
> > You need to call this also for slab to drain the caches and free the pages
> > on the empty list.
> >
> Doesn't the SLAB have a time-based reaper for that?

Yes but it will take a couple of minutes to drain the caches.

Subject: Re: [PATCH v4 24/25] memcg/slub: shrink dead caches
Posted by Glauber Costa on Wed, 25 Jul 2012 18:15:12 GMT
View Forum Message <> Reply to Message

On 07/25/2012 07:23 PM, Christoph Lameter wrote:
> On Fri, 20 Jul 2012, Glauber Costa wrote:
>
>>> This is the same btw in SLAB which keeps objects in per cpu caches and

>>> keeps empty slab pages on special queues.
>>>
>>>> This patch marks all memcg caches as dead. kmem_cache_shrink is called
>>>> for the ones who are not yet dead - this will force internal cache
>>>> reorganization, and then all references to empty pages will be removed.
>>>
>>> You need to call this also for slab to drain the caches and free the pages
>>> on the empty list.
>>>
>> Doesn't the SLAB have a time-based reaper for that?
>
> Yes but it will take a couple of minutes to drain the caches.
>
You might have seen in my last submission that included this in the slab
as well.
