Subject: [PATCH 0/4] kmem memcg proposed core changes
Posted by Glauber Costa on Fri, 08 Jun 2012 09:43:17 GMT
View Forum Message <> Reply to Message

Hello all,

So after thinking a lot about the last round of kmem memcg patches, this
is what I managed to come up with. I am not sending the whole series for two
reasons:

1) It still have a nasty destruction bug, and a slab heisenbug (slub seems to be
   working as flawlessly as before), and I'd like to gather your comments early
   on this approach

2) The rest of the series doesn't change *that* much. Most patches are to some
   extent touched, but it's mainly to adapt to those four, which I consider to
   the the core changes between last series. So you can focus on these, and not
   be distracted by the surrounding churn.

The main difference here is that as suggested by Cristoph, I am hooking at the
page allocator. It is, indeed looser than before. But I still keep objects from
the same cgroup in the same page most of the time. I guarantee that by using the
same dispatch mechanism as before to select a particular per-memcg cache, but I
now assume the process doing the dispatch will be the same doing the page
allocation.

The only situation this does not hold true, is when a task moves cgroup
between those two events. So first of all, this is fixable. One can have a
reaper, a check while moving, a match check after the page is allocated. But
also, this is the kind of loose accounting I don't care too much about, since
this is expected to be a rare event, one I particularly don't care about, and
more importantly, it won't break anything.

Let me know what you people think of this approach. In terms of meddling with
the internals of the caches, it is way less invasive than before.

Glauber Costa (4):
  memcg: kmem controller dispatch infrastructure
  Add a __GFP_SLABMEMCG flag
  don't do __ClearPageSlab before freeing slab page.
  mm: Allocate kernel pages to the right memcg

 include/linux/gfp.h | 4 +-
 include/linux/memcontrol.h | 72 +++++++++
 include/linux/page-flags.h | 2 +-
 include/linux/slub_def.h | 15 +-
 init/Kconfig | 2 +-
 mm/memcontrol.c | 358 ++-

```
 mm/page_alloc.c          |   16 +-
 mm/slab.c                |    9 +-
 mm/slob.c                |    1 -
 mm/slub.c                |    2 +-
 10 files changed, 464 insertions(+), 17 deletions(-)

--
1.7.10.2
```

Subject: [PATCH 1/4] memcg: kmem controller dispatch infrastructure
Posted by Glauber Costa on Fri, 08 Jun 2012 09:43:18 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 |   72 +++++++++
 init/Kconfig               |    2 +-
 mm/memcontrol.c            |  356 ++++++++++++++++++++++++++++++++++++++++++++-
 3 files changed, 428 insertions(+), 2 deletions(-)
```

diff --git a/include/linux/memcontrol.h b/include/linux/memcontrol.h

```
index f93021a..b6acf39 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;
@@ -447,6 +448,16 @@ void mem_cgroup_register_cache(struct mem_cgroup *memcg,
 void mem_cgroup_release_cache(struct kmem_cache *cachep);
 extern char *mem_cgroup_cache_name(struct mem_cgroup *memcg,
        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);
+
+struct kmem_cache *
+__mem_cgroup_get_kmem_cache(struct kmem_cache *cachep, gfp_t gfp);
+
+#define mem_cgroup_kmem_on 1
 #else
 static inline void mem_cgroup_register_cache(struct mem_cgroup *memcg,
          struct kmem_cache *s)
@@ -463,6 +474,67 @@ 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)
+{
+}
+
+#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 (!(gfp & __GFP_SLABMEMCG))
+  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 3764097..45f7ece 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) {
@@ -503,7 +515,14 @@ static struct kmem_cache *kmem_cache_dup(struct mem_cgroup
*memcg,
  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)
@@ -518,9 +537,274 @@ 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);
@@ -4777,7 +5061,11 @@ 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)
@@ -5793,3 +6081,69 @@ 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 2/4] Add a __GFP_SLABMEMCG flag
Posted by Glauber Costa on Fri, 08 Jun 2012 09:43:19 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 | 4 +++-
 1 file changed, 3 insertions(+), 1 deletion(-)

diff --git a/include/linux/gfp.h b/include/linux/gfp.h
index 581e74b..05cfbc2 100644
--- a/include/linux/gfp.h
+++ b/include/linux/gfp.h
@@ -37,6 +37,7 @@ struct vm_area_struct;
 #define ___GFP_NO_KSWAPD 0x400000u
 #define ___GFP_OTHER_NODE 0x800000u
 #define ___GFP_WRITE 0x1000000u
+#define ___GFP_SLABMEMCG 0x2000000u

 /*
  * GFP bitmasks..
@@ -87,6 +88,7 @@ struct vm_area_struct;
 #define __GFP_NO_KSWAPD ((__force gfp_t)___GFP_NO_KSWAPD)
 #define __GFP_OTHER_NODE ((__force gfp_t)___GFP_OTHER_NODE) /* On behalf of other
node */
 #define __GFP_WRITE ((__force gfp_t)___GFP_WRITE) /* Allocator intends to dirty page */
+#define __GFP_SLABMEMCG ((__force gfp_t)___GFP_SLABMEMCG)/* Allocation comes from
a memcg slab */

 /*
  * This may seem redundant, but it's a way of annotating false positives vs.
@@ -94,7 +96,7 @@ struct vm_area_struct;
  */
 #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 3/4] don't do __ClearPageSlab before freeing slab page.
Posted by Glauber Costa on Fri, 08 Jun 2012 09:43:20 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 d7dfd26..66ef370 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -1794,11 +1794,6 @@ 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 ed01be5..a0eeb4a 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 4/4] mm: Allocate kernel pages to the right memcg
Posted by Glauber Costa on Fri, 08 Jun 2012 09:43:21 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 | 15 ++++++++++-----
 mm/memcontrol.c | 2 ++
 mm/page_alloc.c | 13 ++++++++++++-
 mm/slab.c | 4 ++++
 mm/slub.c | 1 +
 6 files changed, 30 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..32aa7a5 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 +227,10 @@ void *__kmalloc(size_t size, gfp_t flags);
 static __always_inline void *
 kmalloc_order(size_t size, gfp_t flags, unsigned int order)
 {
- void *ret = (void *) __get_free_pages(flags | __GFP_COMP, order);
+ void *ret;
+
+ flags |= (__GFP_COMP | __GFP_SLABMEMCG);
+ ret = (void *) __get_free_pages(flags, order);
  kmemleak_alloc(ret, size, 1, flags);
  return ret;
 }
@@ -274,7 +279,7 @@ static __always_inline void *kmalloc(size_t size, gfp_t flags)
    return kmalloc_large(size, flags);

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

```
+   struct kmem_cache *s = kmalloc_slab(flags, size);

    if (!s)
     return ZERO_SIZE_PTR;
@@ -307,7 +312,7 @@ static __always_inline void *kmalloc_node(size_t size, gfp_t flags, int
node)
 {
  if (__builtin_constant_p(size) &&
   size <= SLUB_MAX_SIZE && !(flags & SLUB_DMA)) {
-   struct kmem_cache *s = kmalloc_slab(size);
+   struct kmem_cache *s = kmalloc_slab(flags, size);

   if (!s)
    return ZERO_SIZE_PTR;
diff --git a/mm/memcontrol.c b/mm/memcontrol.c
index 45f7ece..9358140 100644
--- a/mm/memcontrol.c
+++ b/mm/memcontrol.c
@@ -510,6 +510,8 @@ static struct kmem_cache *kmem_cache_dup(struct mem_cgroup
*memcg,
  flags = s->flags & ~(SLAB_PANIC|SLAB_OFF_SLAB);
  new = kmem_cache_create_memcg(memcg, name, s->object_size, s->align,
          flags, 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 66ef370..1b19b34 100644
--- a/mm/slab.c
+++ b/mm/slab.c
@@ -3306,6 +3306,8 @@ __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);

@@ -3391,6 +3393,8 @@ __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 a0eeb4a..6994718 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: Re: [PATCH 2/4] Add a __GFP_SLABMEMCG flag
Posted by Christoph Lameter on Fri, 08 Jun 2012 19:31:07 GMT
View Forum Message <> Reply to Message

On Fri, 8 Jun 2012, Glauber Costa wrote:

```
>   */
>  #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))
```

Please make this conditional on CONFIG_MEMCG or so. The bit can be useful
in particular on 32 bit architectures.

Subject: Re: [PATCH 2/4] Add a __GFP_SLABMEMCG flag
Posted by James Bottomley on Sat, 09 Jun 2012 00:56:56 GMT
View Forum Message <> Reply to Message

On Fri, 2012-06-08 at 14:31 -0500, Christoph Lameter wrote:
> On Fri, 8 Jun 2012, Glauber Costa wrote:
>
> >   */
> > #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))
>

> Please make this conditional on CONFIG_MEMCG or so. The bit can be useful
> in particular on 32 bit architectures.

I really don't think that's at all a good idea. It's asking for trouble when we don't spot we have a flag overlap. It also means that we're trusting the reuser to know that their use case can never clash with CONFIG_MEMGC and I can't think of any configuration where this is possible currently.

I think making the flag define of __GFP_SLABMEMCG conditional might be a reasonable idea so we get a compile failure if anyone tries to use it when !CONFIG_MEMCG.

James

Subject: Re: [PATCH 2/4] Add a __GFP_SLABMEMCG flag
Posted by Glauber Costa on Sat, 09 Jun 2012 08:19:44 GMT
View Forum Message <> Reply to Message

On 06/08/2012 11:31 PM, Christoph Lameter wrote:
> Please make this conditional on CONFIG_MEMCG or so. The bit can be useful
> in particular on 32 bit architectures.
Looking at how __GFP_NOTRACK works - which is also ifdef'd, the bit it uses is skipped if that is not defined, which I believe is a sane thing to do.

Given that, I don't see the point of conditionally defining the memcg bit, It basically means that the only way we can reuse the bit saved is by making a future feature fundamentally incompatible with memcg.

Subject: Re: [PATCH 2/4] Add a __GFP_SLABMEMCG flag
Posted by Glauber Costa on Sat, 09 Jun 2012 08:24:03 GMT
View Forum Message <> Reply to Message

On 06/09/2012 04:56 AM, James Bottomley wrote:
> On Fri, 2012-06-08 at 14:31 -0500, Christoph Lameter wrote:
>> On Fri, 8 Jun 2012, Glauber Costa wrote:
>>
>>> */
>>> #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))
>>

>> Please make this conditional on CONFIG_MEMCG or so. The bit can be useful
>> in particular on 32 bit architectures.
>
> I really don't think that's at all a good idea. It's asking for trouble
> when we don't spot we have a flag overlap. It also means that we're
> trusting the reuser to know that their use case can never clash with
> CONFIG_MEMGC and I can't think of any configuration where this is
> possible currently.
>
> I think making the flag define of __GFP_SLABMEMCG conditional might be a
> reasonable idea so we get a compile failure if anyone tries to use it
> when !CONFIG_MEMCG.
>

Which is also difficult since that's not code that we can BUG or WARN,
but just a number people or and and into their own flags. And it is too
fragile to rely on any given sequence we put here (like -1UL, etc) to
provide predictable enough results to tell someone he is doing it wrong.

A much better approach if we want to protect against that, is to add
code in the page or slab allocator (or both) to ignore and WARN upon
seeing this flag when !memcg.

I'd leave the flag itself alone.

Subject: Re: [PATCH 2/4] Add a __GFP_SLABMEMCG flag
Posted by Christoph Lameter on Mon, 11 Jun 2012 14:24:37 GMT
View Forum Message <> Reply to Message

On Sat, 9 Jun 2012, James Bottomley wrote:

> On Fri, 2012-06-08 at 14:31 -0500, Christoph Lameter wrote:
> > On Fri, 8 Jun 2012, Glauber Costa wrote:
> >
> > >   */
> > > #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))
> >
> > Please make this conditional on CONFIG_MEMCG or so. The bit can be useful
> > in particular on 32 bit architectures.
>
> I really don't think that's at all a good idea. It's asking for trouble
> when we don't spot we have a flag overlap. It also means that we're
> trusting the reuser to know that their use case can never clash with

> CONFIG_MEMGC and I can't think of any configuration where this is
> possible currently.

Flag overlap can be avoided using the same method as we have done with the page flags (which uses an enum). There are other uses of N bits after GFP_BITS_SHIFT. On first look this looks like its 4 right now so we cannot go above 28 on 32 bit platforms. It would also be useful to have that limit in there somehow so that someone modifying the GFP_BITS sees the danger.

> I think making the flag define of __GFP_SLABMEMCG conditional might be a
> reasonable idea so we get a compile failure if anyone tries to use it
> when !CONFIG_MEMCG.

Ok that is another reason to do so.

Subject: Re: [PATCH 2/4] Add a __GFP_SLABMEMCG flag
Posted by James Bottomley on Tue, 12 Jun 2012 14:36:27 GMT
View Forum Message <> Reply to Message

On Mon, 2012-06-11 at 09:24 -0500, Christoph Lameter wrote:
> On Sat, 9 Jun 2012, James Bottomley wrote:
>
> > On Fri, 2012-06-08 at 14:31 -0500, Christoph Lameter wrote:
> > > On Fri, 8 Jun 2012, Glauber Costa wrote:
> > >
> > > > */
> > > > #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))
> > >
> > > Please make this conditional on CONFIG_MEMCG or so. The bit can be useful
> > > in particular on 32 bit architectures.
> >
> > I really don't think that's at all a good idea. It's asking for trouble
> > when we don't spot we have a flag overlap. It also means that we're
> > trusting the reuser to know that their use case can never clash with
> > CONFIG_MEMGC and I can't think of any configuration where this is
> > possible currently.
>
> Flag overlap can be avoided using the same method as we have done with the
> page flags (which uses an enum). There are other uses of N bits after
> GFP_BITS_SHIFT. On first look this looks like its 4 right now so we cannot
> go above 28 on 32 bit platforms. It would also be useful to have that
> limit in there somehow so that someone modifying the GFP_BITS sees the

> danger.

But if there's no possible configuration that can use a flag and depends
on !CONFIG_MEMGC then why bother? The main problem is that unless you
get two configurations which exactly cancel each other and require a GFP
flag, you end up eventually with unbuildable configurations that need
>32 flags.

> > I think making the flag define of __GFP_SLABMEMCG conditional might be a
> > reasonable idea so we get a compile failure if anyone tries to use it
> > when !CONFIG_MEMCG.
>
> Ok that is another reason to do so.

A reason to make it conditional, not a reason to go to the trouble of
making the flags reusable.

James
