Subject: [PATCH 0/4] Kernel memory accounting container (v3)
Posted by Pavel Emelianov on Mon, 17 Sep 2007 12:19:00 GMT
View Forum Message <> Reply to Message

Long time ago we decided to start memory control with the
user memory container. Now this container in -mm tree and
I think we can start with (at least discussion of) the
kmem one.

Changes from v.2:
* introduced generic notifiers for slub. right now there
  are only events, needed by accounting, but this set can
  be extended in the future;
* moved the controller core into separate file, so that
  its extension and/or porting on slAb will look more
  logical;
* fixed this message :).

Changes from v.1:
* fixed Paul's comment about subsystem registration;
* return ERR_PTR from ->create callback, not NULL;
* make container-to-object assignment in rcu-safe section;
* make turning accounting on and off with "1" and "0".

==========

First of all - why do we need this kind of control. The major
"pros" is that kernel memory control protects the system
from DoS attacks by processes that live in container. As our
experience shows many exploits simply do not work in the
container with limited kernel memory.

I can split the kernel memory container into 4 parts:

1. kmalloc-ed objects control
2. vmalloc-ed objects control
3. buddy allocated pages control
4. kmem_cache_alloc-ed objects control

the control of first tree types of objects has one peculiarity:
one need to explicitly point out which allocations he wants to
account and this becomes not-configurable and is to be discussed.

On the other hands such objects as anon_vma-s, file-s, sighangds,
vfsmounts, etc are created by user request always and should
always be accounted. Fortunately they are allocated from their
own caches and thus the whole kmem cache can be accountable.

This is exactly what this patchset does - it adds the ability
to account for the total size of kmem-cache-allocated objects
from specified kmem caches.

This is based on the SLUB allocator, Paul's containers and the
resource counters I made for RSS controller and which are in
-mm tree already.

To play with it, one need to mount the container file system
with -o kmem and then mark some caches as accountable via
/sys/slab/<cache_name>/cache_account.

As I have already told kmalloc caches cannot be accounted easily
so turning the accounting on for them will fail with -EINVAL.
Turning the accounting off is possible only if the cache has
no objects. This is done so because turning accounting off
implies unaccounting of all the objects in the cache, but due
to full-pages in slub are not stored in any lists (usually)
this is impossible to do so, however I'm open for discussion
of how to make this work.

The patches are applicable to the latest Morton's tree.

Thanks,
Pavel

Subject: [PATCH 1/4] Add notification about some major slab events
Posted by Pavel Emelianov on Mon, 17 Sep 2007 12:26:23 GMT
View Forum Message <> Reply to Message

According to Christoph, there are already multiple people who
want to control slab allocations and track memory for various
reasons. So this is an introduction of such a hooks.

The selected method of notification is srcu notifier blocks.
This is selected because the "call" path, i.e. when the
notification is done, is lockless and at the same time
notification handlers can sleep. Neither regular nor atomic
notifiers provide such facilities.

The events tracked are:
1. allocation of an object
2. freeing of an onbject
3. allocation of a new page for objects
4. freeing this page

More events can be added on demand.

The kmem cache marked with SLAB_NOTIFY flag will cause all the
events above to generate notifications. By default no caches
come with this flag.

To preserve the fast-paths and keep the stack from growing
the checks for the flag are made in a separate inline functions
and the actual notification is done in noinline ones.

Hopefully, this looks close to how Christoph sees it :)

Signed-off-by: Pavel Emelyanov <xemul@openvz.org>

```
 include/linux/slab.h     |    1
 include/linux/slub_def.h |   16 +++++++
 mm/slub.c                |  105 ++++++++++++++++++++++++++++++++++++++++++++++-
 3 files changed, 121 insertions(+), 1 deletion(-)

diff --git a/include/linux/slab.h b/include/linux/slab.h
index 3a5bad3..a3bd620 100644
--- a/include/linux/slab.h
+++ b/include/linux/slab.h
@@ -28,6 +28,7 @@
 #define SLAB_DESTROY_BY_RCU 0x00080000UL /* Defer freeing slabs to RCU */
 #define SLAB_MEM_SPREAD  0x00100000UL /* Spread some memory over cpuset */
 #define SLAB_TRACE  0x00200000UL /* Trace allocations and frees */
+#define SLAB_NOTIFY  0x00400000UL /* Notify major events */

 /* The following flags affect the page allocator grouping pages by mobility */
 #define SLAB_RECLAIM_ACCOUNT 0x00020000UL  /* Objects are reclaimable */
diff --git a/include/linux/slub_def.h b/include/linux/slub_def.h
index d65159d..547777e 100644
--- a/include/linux/slub_def.h
+++ b/include/linux/slub_def.h
@@ -200,4 +202,20 @@ static __always_inline void *kmalloc_nod
 }
 #endif

+struct slub_notify_struct {
+ struct kmem_cache *cachep;
+ void *objp;
+ gfp_t gfp;
+};
+
+enum {
+ SLUB_ALLOC,
```

```
+ SLUB_FREE,
+ SLUB_NEWPAGE,
+ SLUB_FREEPAGE,
+};
+
+int slub_register_notifier(struct notifier_block *nb);
+void slub_unregister_notifier(struct notifier_block *nb);
+
 #endif /* _LINUX_SLUB_DEF_H */
diff --git a/mm/slub.c b/mm/slub.c
index 1802645..bfb7c21 100644
--- a/mm/slub.c
+++ b/mm/slub.c
@@ -1013,6 +1013,91 @@ static inline void add_full(struct kmem_
 static inline void kmem_cache_open_debug_check(struct kmem_cache *s) {}
 #define slub_debug 0
 #endif
+
+/*
+ * notifiers
+ */
+
+static struct srcu_notifier_head slub_nb;
+
+static noinline
+int __slub_alloc_notify(int cmd_alloc, int cmd_free, struct kmem_cache *cachep,
+  void *obj, gfp_t gfp)
+{
+ int ret, called;
+ struct slub_notify_struct arg;
+
+ arg.cachep = cachep;
+ arg.objp = obj;
+ arg.gfp = gfp;
+
+ ret = __srcu_notifier_call_chain(&slub_nb, cmd_alloc, &arg,
+   -1, &called);
+ ret = notifier_to_errno(ret);
+
+ if (ret < 0)
+  __srcu_notifier_call_chain(&slub_nb, cmd_free, &arg,
+    called, NULL);
+
+ return ret;
+}
+
+static noinline
+void __slub_free_notify(int cmd, struct kmem_cache *cachep, void *obj)
```

```
+{
+ struct slub_notify_struct arg;
+
+ arg.cachep = cachep;
+ arg.objp = obj;
+ arg.gfp = 0;
+
+ srcu_notifier_call_chain(&slub_nb, cmd, &arg);
+}
+
+int slub_register_notifier(struct notifier_block *nb)
+{
+ return srcu_notifier_chain_register(&slub_nb, nb);
+}
+
+void slub_unregister_notifier(struct notifier_block *nb)
+{
+ srcu_notifier_chain_unregister(&slub_nb, nb);
+}
+
+/*
+ * fastpath hooks
+ */
+
+static inline
+int slub_alloc_notify(struct kmem_cache *cachep, void *obj, gfp_t gfp)
+{
+ return (unlikely(cachep->flags & SLAB_NOTIFY)) ?
+  __slub_alloc_notify(SLUB_ALLOC, SLUB_FREE,
+    cachep, obj, gfp) : 0;
+}
+
+static inline
+void slub_free_notify(struct kmem_cache *cachep, void *obj)
+{
+ if (unlikely(cachep->flags & SLAB_NOTIFY))
+  __slub_free_notify(SLUB_FREE, cachep, obj);
+}
+
+static inline
+int slub_newpage_notify(struct kmem_cache *cachep, struct page *pg, gfp_t gfp)
+{
+ return (unlikely(cachep->flags & SLAB_NOTIFY)) ?
+  __slub_alloc_notify(SLUB_NEWPAGE, SLUB_FREEPAGE,
+    cachep, pg, gfp) : 0;
+}
+
+static inline
```

```
+void slub_freepage_notify(struct kmem_cache *cachep, struct page *pg)
+{
+ if (unlikely(cachep->flags & SLAB_NOTIFY))
+  __slub_free_notify(SLUB_FREEPAGE, cachep, pg);
+}
+
 /*
  * Slab allocation and freeing
  */
@@ -1036,7 +1121,10 @@ static struct page *allocate_slab(struct
   page = alloc_pages_node(node, flags, s->order);

  if (!page)
-  return NULL;
+  goto out;
+
+ if (slub_newpage_notify(s, page, flags) < 0)
+  goto out_free;

  mod_zone_page_state(page_zone(page),
   (s->flags & SLAB_RECLAIM_ACCOUNT) ?
@@ -1044,6 +1132,11 @@ static struct page *allocate_slab(struct
   pages);

  return page;
+
+out_free:
+ __free_pages(page, s->order);
+out:
+ return NULL;
 }

 static void setup_object(struct kmem_cache *s, struct page *page,
@@ -1136,6 +1229,8 @@ static void rcu_free_slab(struct rcu_hea

 static void free_slab(struct kmem_cache *s, struct page *page)
 {
+ slub_freepage_notify(s, page);
+
  if (unlikely(s->flags & SLAB_DESTROY_BY_RCU)) {
   /*
    * RCU free overloads the RCU head over the LRU
@@ -1555,6 +1650,11 @@ static void __always_inline *slab_alloc(
  }
  local_irq_restore(flags);

+ if (object && slub_alloc_notify(s, object, gfpflags) < 0) {
+  kmem_cache_free(s, object);
```

```
+  return NULL;
+ }
+
  if (unlikely((gfpflags & __GFP_ZERO) && object))
   memset(object, 0, c->objsize);

@@ -1651,6 +1751,8 @@ static void __always_inline slab_free(st
  unsigned long flags;
  struct kmem_cache_cpu *c;

+ slub_free_notify(s, x);
+
  local_irq_save(flags);
  debug_check_no_locks_freed(object, s->objsize);
  c = get_cpu_slab(s, smp_processor_id());
@@ -2764,6 +2874,7 @@ void __init kmem_cache_init(void)
  kmem_size = sizeof(struct kmem_cache);
 #endif

+ srcu_init_notifier_head(&slub_nb);

  printk(KERN_INFO "SLUB: Genslabs=%d, HWalign=%d, Order=%d-%d, MinObjects=%d,"
   " CPUs=%d, Nodes=%d\n",
```

Subject: [PATCH 2/4] Switch caches notification dynamically
Posted by Pavel Emelianov on Mon, 17 Sep 2007 12:30:45 GMT
View Forum Message <> Reply to Message

The /sys/slab/<name>/cache_notify attribute controls
whether the cache <name> is to be accounted or not.

For the reasons described before kmalloc caches cannot
be turned on.

By default no caches are accountable. Simply make
  # echo -n 1 > /sys/slab/<name>cache_notify
to turn notification of this cache on.

If we turn accounting on on some cache and this cache
is merged with some other, this "other" will be notified
as well. We can solve this by disabling of cache merging,
but maybe we can do it some other way.

Turning the notification off is possible only when this
cache is empty. The reason for this is that the pages,
that are full of objects are not linked in any list, so
we wouldn't be able to walk these pages and notify others

that these objects are no longer tracked.

Signed-off-by: Pavel Emelyanov <xemul@openvz.org>

---

 mm/slub.c | 45 +++
 1 files changed, 45 insertions(+)

```
diff --git a/mm/slub.c b/mm/slub.c
index 1802645..bfb7c21 100644
--- a/mm/slub.c
+++ b/mm/slub.c
@@ -2338,6 +2440,14 @@ EXPORT_SYMBOL(kmem_cache_destroy);
 struct kmem_cache kmalloc_caches[PAGE_SHIFT] __cacheline_aligned;
 EXPORT_SYMBOL(kmalloc_caches);

+static inline int is_kmalloc_cache(struct kmem_cache *s)
+{
+ int km_idx;
+
+ km_idx = s - kmalloc_caches;
+ return km_idx >= 0 && km_idx < ARRAY_SIZE(kmalloc_caches);
+}
+
 #ifdef CONFIG_ZONE_DMA
 static struct kmem_cache *kmalloc_caches_dma[PAGE_SHIFT];
 #endif
@@ -3753,6 +3874,42 @@ static ssize_t defrag_ratio_store(struct
 SLAB_ATTR(defrag_ratio);
 #endif

+static ssize_t cache_notify_show(struct kmem_cache *s, char *buf)
+{
+ return sprintf(buf, "%d\n", !!(s->flags & SLAB_NOTIFY));
+}
+
+static ssize_t cache_notify_store(struct kmem_cache *s,
+  const char *buf, size_t length)
+{
+ if (buf[0] == '1') {
+  if (is_kmalloc_cache(s))
+   /*
+    * cannot just make these caches accountable
+    */
+   return -EINVAL;
+
+  s->flags |= SLAB_NOTIFY;
```

```
+  return length;
+ }
+
+ if (buf[0] == '0') {
+  if (any_slab_objects(s))
+   /*
+    * we cannot turn this off because of the
+    * full slabs cannot be found in this case
+    */
+   return -EBUSY;
+
+  s->flags &= ~SLAB_NOTIFY;
+  return length;
+ }
+
+ return -EINVAL;
+}
+
+SLAB_ATTR(cache_notify);
+
 static struct attribute * slab_attrs[] = {
  &slab_size_attr.attr,
  &object_size_attr.attr,
@@ -3783,6 +3940,7 @@ static struct attribute * slab_attrs[] =
 #ifdef CONFIG_NUMA
  &defrag_ratio_attr.attr,
 #endif
+ &cache_notify_attr.attr,
  NULL
 };
```

Subject: [PATCH 3/4] Setup the container
Posted by Pavel Emelianov on Mon, 17 Sep 2007 12:33:59 GMT
View Forum Message <> Reply to Message

Attach the controller to the containers. This will work
with the SLUB allocator only. However, if we need I can
port this on SLAB (and maybe SLOB ;)).

This setup is simple and stupid.

Signed-off-by: Pavel Emelyanov <xemul@openvz.org>

```
 include/linux/container_subsys.h |    6 +
 init/Kconfig                     |    6 +
```

```
 mm/Makefile                 |   1
 mm/kmemcontrol.c            | 123 ++++++++++++++++++++++++++++++++++++++++++++++
 4 files changed, 136 insertions(+)

diff --git a/include/linux/container_subsys.h b/include/linux/container_subsys.h
index 81d11c2..9dd90d9 100644
--- a/include/linux/container_subsys.h
+++ b/include/linux/container_subsys.h
@@ -36,3 +36,9 @@ SUBSYS(mem_container)
 #endif

 /* */
+
+#ifdef CONFIG_CONTAINER_KMEM
+SUBSYS(kmem)
+#endif
+
+/* */
diff --git a/init/Kconfig b/init/Kconfig
index 58559ea..d499f15 100644
--- a/init/Kconfig
+++ b/init/Kconfig
@@ -353,6 +353,12 @@ config CONTAINER_MEM_CONT
    Provides a memory controller that manages both page cache and
    RSS memory.

+config CONTAINER_KMEM
+ bool "Kernel memory controller for containers"
+ depends on CONTAINERS && RESOURCE_COUNTERS && SLUB
+ help
+   Provides a kernel memory usage control for containers
+
 config PROC_PID_CPUSET
  bool "Include legacy /proc/<pid>/cpuset file"
  depends on CPUSETS
diff --git a/mm/Makefile b/mm/Makefile
index 6237dd6..1cb7e6d 100644
--- a/mm/Makefile
+++ b/mm/Makefile
@@ -31,4 +31,5 @@ obj-$(CONFIG_MIGRATION) += migrate.o
 obj-$(CONFIG_SMP) += allocpercpu.o
 obj-$(CONFIG_QUICKLIST) += quicklist.o
 obj-$(CONFIG_CONTAINER_MEM_CONT) += memcontrol.o
+obj-$(CONFIG_CONTAINER_KMEM) += kmemcontrol.o

diff --git a/mm/kmemcontrol.c b/mm/kmemcontrol.c
new file mode 100644
index 0000000..637554b
```

```
--- /dev/null
+++ b/mm/kmemcontrol.c
@@ -0,0 +1,123 @@
+/*
+ * kmemcontrol.c - Kernel Memory Controller
+ *
+ * Copyright 2007 OpenVZ SWsoft Inc
+ * Author: Pavel Emelyanov <xemul@openvz.org>
+ *
+ * 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
+ * (at your option) any later version.
+ *
+ * This program is distributed in the hope that it will be useful,
+ * but WITHOUT ANY WARRANTY; without even the implied warranty of
+ * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
+ * GNU General Public License for more details.
+ */
+
+#include <linux/mm.h>
+#include <linux/container.h>
+#include <linux/res_counter.h>
+#include <linux/err.h>
+
+struct kmem_container {
+ struct container_subsys_state css;
+ struct res_counter res;
+};
+
+static inline
+struct kmem_container *css_to_kmem(struct container_subsys_state *css)
+{
+ return container_of(css, struct kmem_container, css);
+}
+
+static inline
+struct kmem_container *container_to_kmem(struct container *cont)
+{
+ return css_to_kmem(container_subsys_state(cont, kmem_subsys_id));
+}
+
+static inline
+struct kmem_container *task_kmem_container(struct task_struct *tsk)
+{
+ return css_to_kmem(task_subsys_state(tsk, kmem_subsys_id));
+}
+
```

```
+/*
+ * containers interface
+ */
+
+static struct kmem_container init_kmem_container;
+
+static struct container_subsys_state *kmem_create(struct container_subsys *ss,
+  struct container *container)
+{
+ struct kmem_container *mem;
+
+ if (unlikely((container->parent) == NULL))
+  mem = &init_kmem_container;
+ else
+  mem = kzalloc(sizeof(struct kmem_container), GFP_KERNEL);
+
+ if (mem == NULL)
+  return ERR_PTR(-ENOMEM);
+
+ res_counter_init(&mem->res);
+ return &mem->css;
+
+}
+
+static void kmem_destroy(struct container_subsys *ss,
+  struct container *container)
+{
+ kfree(container_to_kmem(container));
+}
+
+static ssize_t kmem_container_read(struct container *cont, struct cftype *cft,
+  struct file *file, char __user *userbuf, size_t nbytes,
+  loff_t *ppos)
+{
+ return res_counter_read(&container_to_kmem(cont)->res,
+   cft->private, userbuf, nbytes, ppos);
+}
+
+static ssize_t kmem_container_write(struct container *cont, struct cftype *cft,
+  struct file *file, const char __user *userbuf,
+  size_t nbytes, loff_t *ppos)
+{
+ return res_counter_write(&container_to_kmem(cont)->res,
+   cft->private, userbuf, nbytes, ppos);
+}
+
+static struct cftype kmem_files[] = {
+ {
```

```
+  .name = "usage",
+  .private = RES_USAGE,
+  .read = kmem_container_read,
+ },
+ {
+  .name = "limit",
+  .private = RES_LIMIT,
+  .write = kmem_container_write,
+  .read = kmem_container_read,
+ },
+ {
+  .name = "failcnt",
+  .private = RES_FAILCNT,
+  .read = kmem_container_read,
+ },
+};
+
+static int kmem_populate(struct container_subsys *ss, struct container *cnt)
+{
+ return container_add_files(cnt, ss, kmem_files, ARRAY_SIZE(kmem_files));
+}
+
+struct container_subsys kmem_subsys = {
+ .name = "kmem",
+ .create = kmem_create,
+ .destroy  = kmem_destroy,
+ .populate = kmem_populate,
+ .subsys_id = kmem_subsys_id,
+ .early_init = 1,
+};
```

Subject: [PATCH 4/4] Account for the slub objects
Posted by Pavel Emelianov on Mon, 17 Sep 2007 12:35:59 GMT
View Forum Message <> Reply to Message

The struct page gets an extra pointer (just like it has with
the RSS controller) and this pointer points to the array of
the kmem_container pointers - one for each object stored on
that page itself.

Thus the i'th object on the page is accounted to the container
pointed by the i'th pointer on that array and when the object
is freed we unaccount its size to this particular container,
not the container current task belongs to.

This is done so, because the context objects are freed is most
often not the same as the one this objects was allocated in

(due to RCU and reference counters).

Signed-off-by: Pavel Emelyanov <xemul@openvz.org>

```
 include/linux/mm_types.h |    7 ++
 include/linux/slub_def.h |    2
 mm/kmemcontrol.c         |  126 ++++++
```

```
--- /dev/null
+++ b/mm/kmemcontrol.c
@@ -0,6 +1,9 @@
  * but WITHOUT ANY WARRANTY; without even the implied warranty of
  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  * GNU General Public License for more details.
+ *
+ * Changelog:
+ *   2007 Pavel Emelyanov : Add slub accounting
  */

 #include <linux/mm.h>
@@ -0,2 +123,126 @@
  .subsys_id = kmem_subsys_id,
  .early_init = 1,
 };
+
+/*
+ * slub accounting
+ */
+
+static int kmem_prepare(struct kmem_cache *s, struct page *pg, gfp_t flags)
+{
+ struct kmem_container **ptr;
+
+ ptr = kzalloc(s->objects * sizeof(struct kmem_container *), flags);
+ if (ptr == NULL)
+  return -ENOMEM;
+
+ pg->containers = ptr;
+ return 0;
+}
+
+static void kmem_release(struct kmem_cache *s, struct page *pg)
+{
+ struct kmem_container **ptr;
+
+ ptr = pg->containers;
+ if (ptr == NULL)
+  return;
+
+ kfree(ptr);
+ pg->containers = NULL;
+}
+
+static int kmem_charge(struct kmem_cache *s, void *obj, gfp_t gfp)
+{
+ struct page *pg;
```

```
+ struct kmem_container *cnt;
+ struct kmem_container **obj_container;
+
+ pg = virt_to_head_page(obj);
+ obj_container = pg->containers;
+ if (unlikely(obj_container == NULL)) {
+  /*
+   * turned on after some objects were allocated
+   */
+  if (kmem_prepare(s, pg, gfp) < 0)
+   goto err;
+
+  obj_container = pg->containers;
+ }
+
+ rcu_read_lock();
+ cnt = task_kmem_container(current);
+ if (res_counter_charge(&cnt->res, s->size))
+  goto err_locked;
+
+ css_get(&cnt->css);
+ rcu_read_unlock();
+ obj_container[slab_index(obj, s, page_address(pg))] = cnt;
+ return 0;
+
+err_locked:
+ rcu_read_unlock();
+err:
+ return -ENOMEM;
+}
+
+static void kmem_uncharge(struct kmem_cache *s, void *obj)
+{
+ struct page *pg;
+ struct kmem_container *cnt;
+ struct kmem_container **obj_container;
+
+ pg = virt_to_head_page(obj);
+ obj_container = pg->containers;
+ if (obj_container == NULL)
+  return;
+
+ obj_container += slab_index(obj, s, page_address(pg));
+ cnt = *obj_container;
+ if (cnt == NULL)
+  return;
+
+ res_counter_uncharge(&cnt->res, s->size);
```

```
+ *obj_container = NULL;
+ css_put(&cnt->css);
+}
+
+static int kmem_notify(struct notifier_block *nb, unsigned long cmd, void *arg)
+{
+ int ret;
+ struct slub_notify_struct *ns;
+
+ ns = (struct slub_notify_struct *)arg;
+
+ switch (cmd) {
+ case SLUB_ALLOC:
+  ret = kmem_charge(ns->cachep, ns->objp, ns->gfp);
+  break;
+ case SLUB_FREE:
+  ret = 0;
+  kmem_uncharge(ns->cachep, ns->objp);
+  break;
+ case SLUB_NEWPAGE:
+  ret = kmem_prepare(ns->cachep, ns->objp, ns->gfp);
+  break;
+ case SLUB_FREEPAGE:
+  ret = 0;
+  kmem_release(ns->cachep, ns->objp);
+  break;
+ default:
+  return NOTIFY_DONE;
+ }
+
+ return (ret < 0) ? notifier_from_errno(ret) : NOTIFY_OK;
+}
+
+static struct notifier_block kmem_block = {
+ .notifier_call = kmem_notify,
+};
+
+static int kmem_subsys_register(void)
+{
+ return slub_register_notifier(&kmem_block);
+}
+
+__initcall(kmem_subsys_register);
diff --git a/mm/slub.c b/mm/slub.c
index 1802645..bfb7c21 100644
--- a/mm/slub.c
+++ b/mm/slub.c
@@ -327,7 +327,7 @@ static inline void set_freepointer(struc
```

```
  for (__p = (__free); __p; __p = get_freepointer((__s), __p))

 /* Determine object index from a given position */
-static inline int slab_index(void *p, struct kmem_cache *s, void *addr)
+inline int slab_index(void *p, struct kmem_cache *s, void *addr)
 {
  return (p - addr) / s->size;
 }
@@ -2789,6 +2900,16 @@ static int slab_unmergeable(struct kmem_
  if (s->refcount < 0)
   return 1;

+#ifdef CONFIG_CONTAINER_KMEM
+ /*
+  * many caches that can be accountable are usually merged with
+  * kmalloc caches, which are disabled for accounting for a while
+  */
+
+ if (is_kmalloc_cache(s))
+  return 1;
+#endif
+
  return 0;
 }
```

Subject: Re: [PATCH 4/4] Account for the slub objects
Posted by Dave Hansen on Mon, 17 Sep 2007 16:08:21 GMT
View Forum Message <> Reply to Message

On Mon, 2007-09-17 at 16:35 +0400, Pavel Emelyanov wrote:
> The struct page gets an extra pointer (just like it has with
> the RSS controller) and this pointer points to the array of
> the kmem_container pointers - one for each object stored on
> that page itself.

Can't these at least be unioned so we don't make it any _worse_ when
both are turned on?

-- Dave

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

---

Subject: Re: [PATCH 4/4] Account for the slub objects
Posted by Dave Hansen on Mon, 17 Sep 2007 16:09:28 GMT
View Forum Message <> Reply to Message

On Mon, 2007-09-17 at 16:35 +0400, Pavel Emelyanov wrote:
>
> +
> +	rcu_read_lock();
> +	cnt = task_kmem_container(current);
> +	if (res_counter_charge(&cnt->res, s->size))
> +		goto err_locked;
> +
> +	css_get(&cnt->css);
> +	rcu_read_unlock();
> +	obj_container[slab_index(obj, s, page_address(pg))] = cnt;

You made some effort in the description, but could we get some big fat
comments here about what RCU is doing?

-- Dave

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

Subject: Re: [PATCH 1/4] Add notification about some major slab events
Posted by Christoph Lameter on Mon, 17 Sep 2007 18:25:44 GMT
View Forum Message <> Reply to Message

On Mon, 17 Sep 2007, Pavel Emelyanov wrote:

> @@ -1036,7 +1121,10 @@ static struct page *allocate_slab(struct
> 	page = alloc_pages_node(node, flags, s->order);
>
> 	if (!page)
> -		return NULL;
> +		goto out;
> +
> +	if (slub_newpage_notify(s, page, flags) < 0)
> +		goto out_free;
>
> 	mod_zone_page_state(page_zone(page),
> 		(s->flags & SLAB_RECLAIM_ACCOUNT) ?
> @@ -1044,6 +1132,11 @@ static struct page *allocate_slab(struct
> 		pages);
>

> return page;
> +
> +out_free:
> + __free_pages(page, s->order);
> +out:
> + return NULL;
> }

Ok that looks sane.

> static void setup_object(struct kmem_cache *s, struct page *page,
> @@ -1136,6 +1229,8 @@ static void rcu_free_slab(struct rcu_hea
>
> static void free_slab(struct kmem_cache *s, struct page *page)
> {
> + slub_freepage_notify(s, page);
> +
> if (unlikely(s->flags & SLAB_DESTROY_BY_RCU)) {
> /*
> * RCU free overloads the RCU head over the LRU

Ditto.

> @@ -1555,6 +1650,11 @@ static void __always_inline *slab_alloc(
> }
> local_irq_restore(flags);
>
> + if (object && slub_alloc_notify(s, object, gfpflags) < 0) {
> + kmem_cache_free(s, object);
> + return NULL;
> + }
> +
> if (unlikely((gfpflags & __GFP_ZERO) && object))
> memset(object, 0, c->objsize);
>

Please stay completely out of the fast path. No modifications to
slab_alloc or slab_free please. It is possible to force all allocations of
a particular slab of interest to use the slow path in __slab_alloc (maybe
as a result of the slab page allocation hook returning a certain result
code). See how the SLAB_DEBUG handling does it. You can adapt that and then do the
object checks in __slab_alloc.

Subject: Re: [PATCH 0/4] Kernel memory accounting container (v3)
Posted by Christoph Lameter on Mon, 17 Sep 2007 18:27:11 GMT
View Forum Message <> Reply to Message

On Mon, 17 Sep 2007, Pavel Emelyanov wrote:

> As I have already told kmalloc caches cannot be accounted easily
> so turning the accounting on for them will fail with -EINVAL.
> Turning the accounting off is possible only if the cache has
> no objects. This is done so because turning accounting off
> implies unaccounting of all the objects in the cache, but due
> to full-pages in slub are not stored in any lists (usually)
> this is impossible to do so, however I'm open for discussion
> of how to make this work.

Where can I find more information why is would not be possible to
account kmalloc caches?

Subject: Re: [PATCH 2/4] Switch caches notification dynamically
Posted by Christoph Lameter on Mon, 17 Sep 2007 18:29:33 GMT
View Forum Message <> Reply to Message

On Mon, 17 Sep 2007, Pavel Emelyanov wrote:

> If we turn accounting on on some cache and this cache
> is merged with some other, this "other" will be notified
> as well. We can solve this by disabling of cache merging,
> but maybe we can do it some other way.

You could write a 1 to slub_nomerge during bootup if containers are to
be supported? Once they are merged it is going to be difficult to separate
them again.

Subject: Re: [PATCH 2/4] Switch caches notification dynamically
Posted by Christoph Lameter on Mon, 17 Sep 2007 18:32:42 GMT
View Forum Message <> Reply to Message

On Mon, 17 Sep 2007, Pavel Emelyanov wrote:

> struct kmem_cache kmalloc_caches[PAGE_SHIFT] __cacheline_aligned;
> EXPORT_SYMBOL(kmalloc_caches);
>
> +static inline int is_kmalloc_cache(struct kmem_cache *s)
> +{
> + int km_idx;
> +
> + km_idx = s - kmalloc_caches;
> + return km_idx >= 0 && km_idx < ARRAY_SIZE(kmalloc_caches);
> +}

Could be as simple at

```
 return s > kmalloc_caches && s < kmalloc_caches +
ARRAY_SIZE(kmalloc_caches);
```

```
> + if (buf[0] == '0') {
> +  if (any_slab_objects(s))
> +   /*
> +    * we cannot turn this off because of the
> +    * full slabs cannot be found in this case
> +    */
> +   return -EBUSY;
```

The full slabs can be checked by subtracting the partial slabs from the allocated slabs in the per node structure.

Subject: Re: [PATCH 0/4] Kernel memory accounting container (v3)
Posted by Balbir Singh on Mon, 17 Sep 2007 20:51:03 GMT
View Forum Message <> Reply to Message

Christoph Lameter wrote:
> On Mon, 17 Sep 2007, Pavel Emelyanov wrote:
>
>> As I have already told kmalloc caches cannot be accounted easily
>> so turning the accounting on for them will fail with -EINVAL.
>> Turning the accounting off is possible only if the cache has
>> no objects. This is done so because turning accounting off
>> implies unaccounting of all the objects in the cache, but due
>> to full-pages in slub are not stored in any lists (usually)
>> this is impossible to do so, however I'm open for discussion
>> of how to make this work.
>
> Where can I find more information why is would not be possible to
> account kmalloc caches?
>

Hi, Christoph,

I've wondered the same thing and asked the question. Pavel wrote back to me saying

"The pages that are full of objects are not linked in any list
in kmem_cache so we just cannot find them."

I suspect that SLUB changes this, but I need to look at the
allocator more carefully.

--
Warm Regards,
Balbir Singh
Linux Technology Center
IBM, ISTL

Subject: Re: [PATCH 0/4] Kernel memory accounting container (v3)
Posted by Christoph Lameter on Mon, 17 Sep 2007 21:19:18 GMT
View Forum Message <> Reply to Message

On Tue, 18 Sep 2007, Balbir Singh wrote:

> I've wondered the same thing and asked the question. Pavel wrote
> back to me saying
>
> "The pages that are full of objects are not linked in any list
> in kmem_cache so we just cannot find them."

That is true for any types of slab cache and not restricted to kmalloc
slabs. SLUB can be switched into a mode where it provides these lists
(again at a performance penalty).

But I thought we generate the counters at alloc and free time? So why do
we need to traverse the object lists?

Subject: Re: [PATCH 0/4] Kernel memory accounting container (v3)
Posted by Pavel Emelianov on Tue, 18 Sep 2007 06:25:48 GMT
View Forum Message <> Reply to Message

Christoph Lameter wrote:
> On Mon, 17 Sep 2007, Pavel Emelyanov wrote:
>
>> As I have already told kmalloc caches cannot be accounted easily
>> so turning the accounting on for them will fail with -EINVAL.
>> Turning the accounting off is possible only if the cache has
>> no objects. This is done so because turning accounting off
>> implies unaccounting of all the objects in the cache, but due
>> to full-pages in slub are not stored in any lists (usually)
>> this is impossible to do so, however I'm open for discussion
>> of how to make this work.
>
> Where can I find more information why is would not be possible to

> account kmalloc caches?

It is possible, but the problem is that we want to account only
allocations explicitly caused by the user request. E.g. the
vfsmount name is kmalloc-ed by explicit user request, but such
things as buffer heads are not.

So we have to explicitly specify which calls to kmalloc() do we
wand to be accounted and which we do not by passing additional
flag (GFP_ACCT?) to kmalloc, but this is another patch.

Yet again - as soon as we agree on the implementation of kmem
caches accounting, I will proceed with working on kmalloc, vmalloc
and buddy allocator.

>

Thanks,
Pavel

Subject: Re: [PATCH 4/4] Account for the slub objects
Posted by Pavel Emelianov on Tue, 18 Sep 2007 06:27:31 GMT
View Forum Message <> Reply to Message

Dave Hansen wrote:
> On Mon, 2007-09-17 at 16:35 +0400, Pavel Emelyanov wrote:
>> The struct page gets an extra pointer (just like it has with
>> the RSS controller) and this pointer points to the array of
>> the kmem_container pointers - one for each object stored on
>> that page itself.
>
> Can't these at least be unioned so we don't make it any _worse_ when
> both are turned on?

Your comment makes me suspect, that you don't look at the patches
at all :(

They ARE unioned.

> -- Dave
>
>

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

Subject: Re: [PATCH 4/4] Account for the slub objects
Posted by Pavel Emelianov on Tue, 18 Sep 2007 06:28:26 GMT
View Forum Message <> Reply to Message

Dave Hansen wrote:
> On Mon, 2007-09-17 at 16:35 +0400, Pavel Emelyanov wrote:
>> +
>> +	rcu_read_lock();
>> +	cnt = task_kmem_container(current);
>> +	if (res_counter_charge(&cnt->res, s->size))
>> +		goto err_locked;
>> +
>> +	css_get(&cnt->css);
>> +	rcu_read_unlock();
>> +	obj_container[slab_index(obj, s, page_address(pg))] = cnt;
>
> You made some effort in the description, but could we get some big fat
> comments here about what RCU is doing?

No big fat comment here - this all is containers API.

> -- Dave
>
>

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

Subject: Re: [PATCH 2/4] Switch caches notification dynamically
Posted by Pavel Emelianov on Tue, 18 Sep 2007 06:51:16 GMT
View Forum Message <> Reply to Message

Christoph Lameter wrote:
> On Mon, 17 Sep 2007, Pavel Emelyanov wrote:
>
>> If we turn accounting on on some cache and this cache
>> is merged with some other, this "other" will be notified
>> as well. We can solve this by disabling of cache merging,
>> but maybe we can do it some other way.
>
> You could write a 1 to slub_nomerge during bootup if containers are to
> be supported? Once they are merged it is going to be difficult to separate
> them again.

Yes. I also thought that disabling the merge is the only way to

handle this case. Thanks.

Subject: Re: [PATCH 2/4] Switch caches notification dynamically
Posted by Pavel Emelianov on Tue, 18 Sep 2007 06:54:08 GMT
View Forum Message <> Reply to Message

Christoph Lameter wrote:
> On Mon, 17 Sep 2007, Pavel Emelyanov wrote:
>
>> struct kmem_cache kmalloc_caches[PAGE_SHIFT] __cacheline_aligned;
>> EXPORT_SYMBOL(kmalloc_caches);
>>
>> +static inline int is_kmalloc_cache(struct kmem_cache *s)
>> +{
>> + int km_idx;
>> +
>> + km_idx = s - kmalloc_caches;
>> + return km_idx >= 0 && km_idx < ARRAY_SIZE(kmalloc_caches);
>> +}
>
>
> Could be as simple at
>
> return s > kmalloc_caches && s < kmalloc_caches +
> ARRAY_SIZE(kmalloc_caches);
>
>
>> + if (buf[0] == '0') {
>> + if (any_slab_objects(s))
>> + /*
>> + * we cannot turn this off because of the
>> + * full slabs cannot be found in this case
>> + */
>> + return -EBUSY;
>
> The full slabs can be checked by subtracting the partial slabs from the
> allocated slabs in the per node structure.

No no! This is not that I meant here. This is just like the redzoning
turning on/off dynamically.

I meant that we cannot find the pages that are full of objects to notify
others that these ones are no longer tracked. I know that we can do
it by tracking these pages with some performance penalty, but does it
worth having the ability to turn notifications off by the cost of the
performance degradation?

Thanks,
Pavel

Subject: Re: [PATCH 0/4] Kernel memory accounting container (v3)
Posted by Pavel Emelianov on Tue, 18 Sep 2007 06:56:21 GMT
View Forum Message <> Reply to Message

Christoph Lameter wrote:
> On Tue, 18 Sep 2007, Balbir Singh wrote:
>
>> I've wondered the same thing and asked the question. Pavel wrote
>> back to me saying
>>
>> "The pages that are full of objects are not linked in any list
>> in kmem_cache so we just cannot find them."
>
> That is true for any types of slab cache and not restricted to kmalloc
> slabs. SLUB can be switched into a mode where it provides these lists
> (again at a performance penalty).
>
> But I thought we generate the counters at alloc and free time? So why do
> we need to traverse the object lists?

When we make echo 0 > /sys/slab/xxx/cache_notify we want all the objects
to be unaccounted back immediately. Even __free_slab() won't catch this
because the SLAB_NOTIFY flag will be turned off for this cache. So we
have to walk all the objects and unaccount them.

Subject: Re: [PATCH 1/4] Add notification about some major slab events
Posted by Pavel Emelianov on Tue, 18 Sep 2007 08:03:43 GMT
View Forum Message <> Reply to Message

Christoph Lameter wrote:
> On Mon, 17 Sep 2007, Pavel Emelyanov wrote:
>
>> @@ -1036,7 +1121,10 @@ static struct page *allocate_slab(struct
>> page = alloc_pages_node(node, flags, s->order);
>>
>> if (!page)
>> - return NULL;
>> + goto out;
>> +
>> + if (slub_newpage_notify(s, page, flags) < 0)
>> + goto out_free;
>>

>>   mod_zone_page_state(page_zone(page),
>>   (s->flags & SLAB_RECLAIM_ACCOUNT) ?
>> @@ -1044,6 +1132,11 @@ static struct page *allocate_slab(struct
>>   pages);
>>
>>   return page;
>> +
>> +out_free:
>> + __free_pages(page, s->order);
>> +out:
>> + return NULL;
>> }
>
> Ok that looks sane.
>
>> static void setup_object(struct kmem_cache *s, struct page *page,
>> @@ -1136,6 +1229,8 @@ static void rcu_free_slab(struct rcu_hea
>>
>> static void free_slab(struct kmem_cache *s, struct page *page)
>> {
>> + slub_freepage_notify(s, page);
>> +
>>   if (unlikely(s->flags & SLAB_DESTROY_BY_RCU)) {
>>   /*
>>   * RCU free overloads the RCU head over the LRU
>
> Ditto.
>
>> @@ -1555,6 +1650,11 @@ static void __always_inline *slab_alloc(
>>   }
>>   local_irq_restore(flags);
>>
>> + if (object && slub_alloc_notify(s, object, gfpflags) < 0) {
>> + kmem_cache_free(s, object);
>> + return NULL;
>> + }
>> +
>>   if (unlikely((gfpflags & __GFP_ZERO) && object))
>>   memset(object, 0, c->objsize);
>>
>
> Please stay completely out of the fast path. No modifications to
> slab_alloc or slab_free please. It is possible to force all allocations of
> a particular slab of interest to use the slow path in __slab_alloc (maybe
> as a result of the slab page allocation hook returning a certain result
> code). See how the SLAB_DEBUG handling does it. You can adapt that and then do the
> object checks in __slab_alloc.

That's true, but:
1. we perform only a flag check on a fast path
2. currently we cannot force the freeing of an object to go _always_
   through the slow __slab_free(), and thus the following situation is
   possible:
     a. container A allocates an object and this object is
        accounted to it
     b. the object is freed and gets into lockless freelist (but
        stays accounted to A)
     c. container C allocates this object from the freelist
        and thus get unaccounted amount of memory
  this discrepancy can grow up infinitely. Sure, we can mark some caches to
  go through the slow path even on freeing the objects, but isn't it the
  same as checking for SLAB_NOTIFY on fast paths?

Maybe it's worth having the notifiers under config option, so that those
who don't need this won't suffer at all?

Thanks,
Pavel

Subject: Re: [PATCH 1/4] Add notification about some major slab events
Posted by Christoph Lameter on Tue, 18 Sep 2007 19:35:07 GMT
View Forum Message <> Reply to Message

On Tue, 18 Sep 2007, Pavel Emelyanov wrote:

> > Please stay completely out of the fast path. No modifications to
> > slab_alloc or slab_free please. It is possible to force all allocations of
> > a particular slab of interest to use the slow path in __slab_alloc (maybe
> > as a result of the slab page allocation hook returning a certain result
> > code). See how the SLAB_DEBUG handling does it. You can adapt that and then do the
> > object checks in __slab_alloc.
>
> That's true, but:
> 1. we perform only a flag check on a fast path

This is still going to slow down everyone else and I still think there is
no need to do that.

> 2. currently we cannot force the freeing of an object to go _always_
>   through the slow __slab_free(), and thus the following situation is
>   possible:
>   a. container A allocates an object and this object is
>   accounted to it

At that point you could mark the slab as a slow slab by setting

SLAB_DEBUG() so we always take the slow path for this slab.

> b. the object is freed and gets into lockless freelist (but
> stays accounted to A)

B wont work if SLAB_DEBUG() is set. The fastpath is then disabled.

> c. container C allocates this object from the freelist
> and thus get unaccounted amount of memory

> this discrepancy can grow up infinitely. Sure, we can mark some caches to
> go through the slow path even on freeing the objects, but isn't it the
> same as checking for SLAB_NOTIFY on fast paths?

The other caches will then still perform up to speed.

> Maybe it's worth having the notifiers under config option, so that those
> who don't need this won't suffer at all?

I think you would want the container functionality to be available in distros. They may make the choice whether to enable the container functionality based on its impact. It is good if we can stash it away so that there is a negligible performance impact if its compiled in but off.

Subject: Re: [PATCH 2/4] Switch caches notification dynamically
Posted by Christoph Lameter on Tue, 18 Sep 2007 19:36:23 GMT
View Forum Message <> Reply to Message

On Tue, 18 Sep 2007, Pavel Emelyanov wrote:

> I meant that we cannot find the pages that are full of objects to notify
> others that these ones are no longer tracked. I know that we can do
> it by tracking these pages with some performance penalty, but does it
> worth having the ability to turn notifications off by the cost of the
> performance degradation?

Not sure. That may be your call.

Subject: Re: [PATCH 0/4] Kernel memory accounting container (v3)
Posted by Christoph Lameter on Tue, 18 Sep 2007 19:37:44 GMT
View Forum Message <> Reply to Message

On Tue, 18 Sep 2007, Pavel Emelyanov wrote:

> > Where can I find more information why is would not be possible to

> > account kmalloc caches?
>
> It is possible, but the problem is that we want to account only
> allocations explicitly caused by the user request. E.g. the
> vfsmount name is kmalloc-ed by explicit user request, but such
> things as buffer heads are not.
>
> So we have to explicitly specify which calls to kmalloc() do we
> wand to be accounted and which we do not by passing additional
> flag (GFP_ACCT?) to kmalloc, but this is another patch.
>
> Yet again - as soon as we agree on the implementation of kmem
> caches accounting, I will proceed with working on kmalloc, vmalloc
> and buddy allocator.

Oh gosh..... Lots of complications in the allocator paths.

Subject: Re: [PATCH 1/4] Add notification about some major slab events
Posted by Pavel Emelianov on Wed, 19 Sep 2007 10:08:32 GMT
View Forum Message <> Reply to Message

[snip]

>> @@ -1555,6 +1650,11 @@ static void __always_inline *slab_alloc(
>> }
>> local_irq_restore(flags);
>>
>> + if (object && slub_alloc_notify(s, object, gfpflags) < 0) {
>> + kmem_cache_free(s, object);
>> + return NULL;
>> + }
>> +
>> if (unlikely((gfpflags & __GFP_ZERO) && object))
>> memset(object, 0, c->objsize);
>>
>
> Please stay completely out of the fast path. No modifications to
> slab_alloc or slab_free please. It is possible to force all allocations of
> a particular slab of interest to use the slow path in __slab_alloc (maybe
> as a result of the slab page allocation hook returning a certain result
> code). See how the SLAB_DEBUG handling does it. You can adapt that and then do the
> object checks in __slab_alloc.

I have run the kernbench test on the kernels with a) containers support
and b) containers and kmem accounting support but (!) turned off. The
results are:
a) b)

Elapsed Time 768.500000 767.050000
User Time 679.360000 679.240000
System Time 87.020000 86.950000
Percent CPU 99.000000 99.000000
Context Switches 376891.000000 375407.000000
Sleeps 385377.000000 385426.000000

The test run was kernbench -n 1 -o 4 -M, the node is i386
DualCore Xeon 3.2GHz with 2Gb of RAM.

so the fast path is still fast, and we have two ways:
1. we keep the checks on the fastpath and have 0 overhead for
   unaccounted caches and some overhead for accounted;
2. we move the checks into the slow one and have 0 overhead for
   unaccounted caches and huge overhead for accounted.

I admit that I messed something, so shall I measure some
other activity or use another HW?

Thanks,
Pavel

Subject: Re: [PATCH 1/4] Add notification about some major slab events
Posted by Christoph Lameter on Wed, 19 Sep 2007 17:45:32 GMT
View Forum Message <> Reply to Message

On Wed, 19 Sep 2007, Pavel Emelyanov wrote:

> so the fast path is still fast, and we have two ways:
> 1. we keep the checks on the fastpath and have 0 overhead for
>   unaccounted caches and some overhead for accounted;

This stuff accumulates. I have a bad experience from SLAB. We are counting
cycle counts and cachelines touched in the fastpath and this is going to
add to them.

> 2. we move the checks into the slow one and have 0 overhead for
>   unaccounted caches and huge overhead for accounted.

Huge? Its not that huge.

> I admit that I messed something, so shall I measure some
> other activity or use another HW?

You could use this module to test the cycles in the fastpath:

```
/* test-slub.c
 */


#include <linux/jiffies.h>
#include <linux/compiler.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/calc64.h>
#include <asm/timex.h>
#include <asm/system.h>

#define TEST_COUNT 10000

#define PARALLEL

#ifdef PARALLEL
#include <linux/completion.h>
#include <linux/sched.h>
#include <linux/workqueue.h>
#include <linux/kthread.h>

struct test_struct {
 struct task_struct *task;
 int cpu;
 int size;
 int count;
 void **v;
 void (*test_p1)(struct test_struct *);
 void (*test_p2)(struct test_struct *);
 unsigned long start;
 unsigned long stop1;
 unsigned long stop2;
} test[NR_CPUS];

/*
 * Allocate TEST_COUNT objects and later free them all again
 */
static void kmalloc_alloc_then_free_test_p1(struct test_struct *t)
{
 int i;

 for (i = 0; i < t->count; i++)
  t->v[i] = kmalloc(t->size, GFP_KERNEL);
}

static void kmalloc_alloc_then_free_test_p2(struct test_struct *t)
{
```

```
 int i;

 for (i = 0; i < t->count; i++)
  kfree(t->v[i]);
}

/*
 * Allocate TEST_COUNT objects. Free them immediately.
 */
static void kmalloc_alloc_free_test_p1(struct test_struct *t)
{
 int i;

 for (i = 0; i < TEST_COUNT; i++)
  kfree(kmalloc(t->size, GFP_KERNEL));
}

static atomic_t tests_running;
static DECLARE_COMPLETION(completion);
static int started;

static int test_func(void *private)
{
 struct test_struct *t = private;
 cpumask_t newmask = CPU_MASK_NONE;

        cpu_set(t->cpu, newmask);
        set_cpus_allowed(current, newmask);
 t->v = kmalloc(t->count * sizeof(void *), GFP_KERNEL);

 atomic_inc(&tests_running);
 wait_for_completion(&completion);
 t->start = get_cycles();
 t->test_p1(t);
 t->stop1 = get_cycles();
 if (t->test_p2)
  t->test_p2(t);
 t->stop2 = get_cycles();
 kfree(t->v);
 atomic_dec(&tests_running);
 set_current_state(TASK_UNINTERRUPTIBLE);
 schedule();
 return 0;
}

static void do_concurrent_test(void (*p1)(struct test_struct *),
  void (*p2)(struct test_struct *),
  int size, const char *name)
```

```
{
 int cpu;
 unsigned long time1 = 0;
 unsigned long time2 = 0;
 unsigned long sum1 = 0;
 unsigned long sum2 = 0;

 atomic_set(&tests_running, 0);
 started = 0;
 init_completion(&completion);

 for_each_online_cpu(cpu) {
  struct test_struct *t = &test[cpu];

  t->cpu = cpu;
  t->count = TEST_COUNT;
  t->test_p1 = p1;
  t->test_p2 = p2;
  t->size = size;
  t->task = kthread_run(test_func, t, "test%d", cpu);
  if (IS_ERR(t->task)) {
   printk("Failed to start test func\n");
   return;
  }
 }

 /* Wait till all processes are running */
 while (atomic_read(&tests_running) < num_online_cpus()) {
  set_current_state(TASK_UNINTERRUPTIBLE);
  schedule_timeout(10);
 }
 complete_all(&completion);
 while (atomic_read(&tests_running)) {
  set_current_state(TASK_UNINTERRUPTIBLE);
  schedule_timeout(10);
 }

 for_each_online_cpu(cpu)
  kthread_stop(test[cpu].task);

 printk(KERN_ALERT "%s(%d):", name, size);
 for_each_online_cpu(cpu) {
  struct test_struct *t = &test[cpu];

  time1 = t->stop1 - t->start;
  time2 = t->stop2 - t->stop1;
  sum1 += time1;
  sum2 += time2;
```

```
  printk(" %d=%lu", cpu, time1 / TEST_COUNT);
  if (p2)
   printk("/%lu", time2 / TEST_COUNT);
 }
 printk(" Average=%lu", sum1 / num_online_cpus() / TEST_COUNT);
 if (p2)
  printk("/%lu", sum2 / num_online_cpus() / TEST_COUNT);
 printk("\n");
 schedule_timeout(200);
}
#endif

static int slub_test_init(void)
{
 void **v = kmalloc(TEST_COUNT * sizeof(void *), GFP_KERNEL);
 unsigned int i;
 cycles_t time1, time2, time;
 long rem;
 int size;

 printk(KERN_ALERT "test init\n");

 printk(KERN_ALERT "Single thread testing\n");
 printk(KERN_ALERT "=====================\n");
 printk(KERN_ALERT "1. Kmalloc: Repeatedly allocate then free test\n");
 for (size = 8; size <= PAGE_SIZE << 2; size <<= 1) {
  time1 = get_cycles();
  for (i = 0; i < TEST_COUNT; i++) {
   v[i] = kmalloc(size, GFP_KERNEL);
  }
  time2 = get_cycles();
  time = time2 - time1;

  printk(KERN_ALERT "%i times kmalloc(%d) ", i, size);
  time = div_long_long_rem(time, TEST_COUNT, &rem);
  printk("-> %llu cycles ", time);

  time1 = get_cycles();
  for (i = 0; i < TEST_COUNT; i++) {
   kfree(v[i]);
  }
  time2 = get_cycles();
  time = time2 - time1;

  printk("kfree ");
  time = div_long_long_rem(time, TEST_COUNT, &rem);
  printk("-> %llu cycles\n", time);
 }
```

```
 printk(KERN_ALERT "2. Kmalloc: alloc/free test\n");
 for (size = 8; size <= PAGE_SIZE << 2; size <<= 1) {
  time1 = get_cycles();
  for (i = 0; i < TEST_COUNT; i++) {
   kfree(kmalloc(size, GFP_KERNEL));
  }
  time2 = get_cycles();
  time = time2 - time1;

  printk(KERN_ALERT "%i times kmalloc(%d)/kfree ", i, size);
  time = div_long_long_rem(time, TEST_COUNT, &rem);
  printk("-> %llu cycles\n", time);
 }
 kfree(v);
#ifdef PARALLEL
 printk(KERN_INFO "Concurrent allocs\n");
 printk(KERN_INFO "==================\n");
 for (i = 3; i <= PAGE_SHIFT; i++) {
  do_concurrent_test(kmalloc_alloc_then_free_test_p1,
   kmalloc_alloc_then_free_test_p2,
   1 << i, "Kmalloc N*alloc N*free");
 }
 for (i = 3; i <= PAGE_SHIFT; i++) {
  do_concurrent_test(kmalloc_alloc_free_test_p1, NULL,
   1 << i, "Kmalloc N*(alloc free)");
 }
#endif

 return -EAGAIN; /* Fail will directly unload the module */
}

static void slub_test_exit(void)
{
 printk(KERN_ALERT "test exit\n");
}

module_init(slub_test_init)
module_exit(slub_test_exit)

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Mathieu Desnoyers");
MODULE_DESCRIPTION("SLUB test");
```