Subject: [RFC][PATCH 0/3] Kernel memory accounting container (v2)
Posted by Pavel Emelianov on Thu, 13 Sep 2007 09:11:35 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.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.

I know it's maybe too late, since some of you may be preparing
for the Summit or LinixConf, but I think that we can go on
discussing these on LinuxConf.

The patches are applicable to the latest Morton's tree (that
without the RSS controll) with the resource counters patch
Andrew committed recently.

I've made some minimal testing for that and the similar code
(without the containers interface but with the kmalloc
accounting) is already in our 2.6.22 OpenVZ tree, so testing
is going on.

Thanks,
Pavel

Subject: [RFC][PATCH 1/3] Setup the kmem container
Posted by Pavel Emelianov on Thu, 13 Sep 2007 09:13:11 GMT
View Forum Message <> Reply to Message

[PATCH 1/3] Setup the kmem container

Attach the controller to the containers. This can work
with the SLUB allocator only. However, if we need I can
port this on SLAB (and even 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/slub.c                        |  110 ++++++++++++++++++++++++++++++++++++++++
```

```
 3 files changed, 122 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/slub.c b/mm/slub.c
index 1802645..16da4d4 100644
--- a/mm/slub.c
+++ b/mm/slub.c
@@ -21,6 +21,12 @@
 #include <linux/ctype.h>
 #include <linux/kallsyms.h>

+#ifdef CONFIG_CONTAINER_KMEM
+#include <linux/container.h>
+#include <linux/res_counter.h>
+#include <linux/err.h>
+#endif
+
 /*
```

```
 * Lock order:
 *   1. slab_lock(page)
@@ -4011,3 +4016,107 @@ static int __init slab_sysfs_init(void)

 __initcall(slab_sysfs_init);
 #endif
+
+#ifdef CONFIG_CONTAINER_KMEM
+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,
+};
+#endif
```

Subject: [RFC][PATCH 2/3] The accounting hooks and core
Posted by Pavel Emelianov on Thu, 13 Sep 2007 09:14:40 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).

Kmem cache marked as SLAB_CHARGE will perform the accounting.

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

```
 include/linux/mm_types.h |   12 +++
 include/linux/slab.h     |    1
 mm/slub.c                |  165 ++++++++++++++++++++++++++++++++++++++++++++++++-
 3 files changed, 176 insertions(+), 2 deletions(-)


diff --git a/include/linux/mm_types.h b/include/linux/mm_types.h
index 48df4b4..67e8ea4 100644
```

```
--- a/include/linux/mm_types.h
+++ b/include/linux/mm_types.h
@@ -83,9 +83,19 @@ struct page {
  void *virtual;   /* Kernel virtual address (NULL if
        not kmapped, ie. highmem) */
 #endif /* WANT_PAGE_VIRTUAL */
+
+ /*
+  * one page cannot be mapped to the userspace and be
+  * allocated for slub at the same time
+  */
+ union {
 #ifdef CONFIG_CONTAINER_MEM_CONT
- unsigned long page_container;
+  unsigned long page_container;
+#endif
+#ifdef CONFIG_CONTAINER_KMEM
+  struct kmem_container **containers;
 #endif
+ };
 #ifdef CONFIG_PAGE_OWNER
  int order;
  unsigned int gfp_mask;
diff --git a/include/linux/slab.h b/include/linux/slab.h
index 3a5bad3..cd7d50d 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_CHARGE  0x00400000UL /* Charge allocations */

 /* The following flags affect the page allocator grouping pages by mobility */
 #define SLAB_RECLAIM_ACCOUNT 0x00020000UL  /* Objects are reclaimable */
diff --git a/mm/slub.c b/mm/slub.c
index 16da4d4..113df81 100644
--- a/mm/slub.c
+++ b/mm/slub.c
@@ -1018,6 +1018,73 @@ static inline void add_full(struct kmem_
 static inline void kmem_cache_open_debug_check(struct kmem_cache *s) {}
 #define slub_debug 0
 #endif
+
+#ifdef CONFIG_CONTAINER_KMEM
+/*
+ * Fast path stubs
+ */
```

```
+
+static int __kmem_charge(struct kmem_cache *s, void *obj, gfp_t flags);
+static inline
+int kmem_charge(struct kmem_cache *s, void *obj, gfp_t flags)
+{
+ return (s->flags & SLAB_CHARGE) ? __kmem_charge(s, obj, flags) : 0;
+}
+
+static void __kmem_uncharge(struct kmem_cache *s, void *obj);
+static inline
+void kmem_uncharge(struct kmem_cache *s, void *obj)
+{
+ if (s->flags & SLAB_CHARGE)
+  __kmem_uncharge(s, obj);
+}
+
+static int __kmem_prepare(struct kmem_cache *s, struct page *pg, gfp_t flags);
+static inline
+int kmem_prepare(struct kmem_cache *s, struct page *pg, gfp_t flags)
+{
+ return (s->flags & SLAB_CHARGE) ? __kmem_prepare(s, pg, flags) : 0;
+}
+
+static void __kmem_release(struct kmem_cache *s, struct page *pg);
+static inline
+void kmem_release(struct kmem_cache *s, struct page *pg)
+{
+ if (s->flags & SLAB_CHARGE)
+  __kmem_release(s, pg);
+}
+
+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);
+}
+#else
+static inline
+int kmem_charge(struct kmem_cache *s, void *obj, gfp_t flags)
+{
+ return 0;
+}
+
+static inline
+void kmem_uncharge(struct kmem_cache *s, void *obj)
+{
```

```
+}
+
+static inline
+int kmem_prepare(struct kmem_cache *s, struct page *pg, gfp_t flags)
+{
+ return 0;
+}
+
+static inline
+void kmem_release(struct kmem_cache *s, struct page *pg)
+{
+}
+#endif
+
 /*
  * Slab allocation and freeing
  */
@@ -1041,7 +1108,10 @@ static struct page *allocate_slab(struct
   page = alloc_pages_node(node, flags, s->order);

  if (!page)
-  return NULL;
+  goto err_page;
+
+ if (kmem_prepare(s, page, flags) < 0)
+  goto err_prep;

  mod_zone_page_state(page_zone(page),
   (s->flags & SLAB_RECLAIM_ACCOUNT) ?
@@ -1049,6 +1119,11 @@ static struct page *allocate_slab(struct
   pages);

  return page;
+
+err_prep:
+ __free_pages(page, s->order);
+err_page:
+ return NULL;
 }

 static void setup_object(struct kmem_cache *s, struct page *page,
@@ -1141,6 +1216,8 @@ static void rcu_free_slab(struct rcu_hea

 static void free_slab(struct kmem_cache *s, struct page *page)
 {
+ kmem_release(s, page);
+
  if (unlikely(s->flags & SLAB_DESTROY_BY_RCU)) {
```

```
   /*
    * RCU free overloads the RCU head over the LRU
@@ -1560,6 +1637,11 @@ static void __always_inline *slab_alloc(
  }
  local_irq_restore(flags);

+ if (object && kmem_charge(s, object, gfpflags) < 0) {
+  kmem_cache_free(s, object);
+  return NULL;
+ }
+
  if (unlikely((gfpflags & __GFP_ZERO) && object))
   memset(object, 0, c->objsize);

@@ -1656,6 +1738,8 @@ static void __always_inline slab_free(st
  unsigned long flags;
  struct kmem_cache_cpu *c;

+ kmem_uncharge(s, x);
+
  local_irq_save(flags);
  debug_check_no_locks_freed(object, s->objsize);
  c = get_cpu_slab(s, smp_processor_id());
@@ -4041,6 +4125,85 @@ struct kmem_container *task_kmem_contain
  return css_to_kmem(task_subsys_state(tsk, kmem_subsys_id));
 }

+static int __kmem_charge(struct kmem_cache *s, void *obj, gfp_t flags)
+{
+ 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, flags) < 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_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;
+}
+
 /*
  * containers interface
  */
```

Subject: [RFC][PATCH 3/3] Tune caches to be accountable or not
Posted by Pavel Emelianov on Thu, 13 Sep 2007 09:16:05 GMT
View Forum Message <> Reply to Message

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

For the reasons described in the zeroth letter kmalloc
caches are excluded and are not allowed to be merged.

By default no caches are accountable. Simply make
  # echo -n 1 > /sys/slab/<name>cache_account
to turn accounting on.

Other caches can be accountable, but if we turn accounting
on on some cache and this cache is merged with some other,
this "other" will be accountable as well. We can solve this
by disabling of cache merging, but I'd prefer to know
Christoph's opinion first.

Turning the accounting off is possible only when this cache
is empty.

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

---

 mm/slub.c | 51 ++
 1 files changed, 51 insertions(+)

diff --git a/mm/slub.c b/mm/slub.c
index 113df81..baf8da4 100644
--- a/mm/slub.c

```
+++ b/mm/slub.c
@@ -2878,6 +2878,16 @@ 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;
 }

@@ -3842,6 +3852,44 @@ static ssize_t defrag_ratio_store(struct
 SLAB_ATTR(defrag_ratio);
 #endif

+#ifdef CONFIG_CONTAINER_KMEM
+static ssize_t cache_account_show(struct kmem_cache *s, char *buf)
+{
+ return sprintf(buf, "%d\n", !!(s->flags & SLAB_CHARGE));
+}
+
+static ssize_t cache_account_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_CHARGE;
+  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_CHARGE;
+  return length;
+ }
+
+ return -EINVAL;
+}
+
+SLAB_ATTR(cache_account);
+#endif
+
 static struct attribute * slab_attrs[] = {
  &slab_size_attr.attr,
  &object_size_attr.attr,
@@ -3872,6 +3920,9 @@ static struct attribute * slab_attrs[] =
 #ifdef CONFIG_NUMA
  &defrag_ratio_attr.attr,
 #endif
+#ifdef CONFIG_CONTAINER_KMEM
+ &cache_account_attr.attr,
+#endif
  NULL
 };
```

Subject: Re: [RFC][PATCH 0/3] Kernel memory accounting container (v2)
Posted by KAMEZAWA Hiroyuki on Thu, 13 Sep 2007 10:19:50 GMT
View Forum Message <> Reply to Message

On Thu, 13 Sep 2007 13:11:35 +0400
Pavel Emelyanov <xemul@openvz.org> wrote:

> 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
>
<snip>
> 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.
>
Hmm, how can we know "How many kmem will we need ?" in precise per-object
style ? Is this useful ?

Following kind of limitation of user friendly params is bad ?

 - # of file handles
 - # of tasks
 - # of sockets/ connections / packets
 - # of posix IPC related things
 - and other sources of DoS.

Thanks,
-Kame

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

---

Subject: Re: [RFC][PATCH 0/3] Kernel memory accounting container (v2)
Posted by Balbir Singh on Thu, 13 Sep 2007 10:46:56 GMT
View Forum Message <> Reply to Message

Pavel Emelyanov wrote:
> 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.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.
>

Does this mean that the kernel memory container will have a dependency
on SLUB and it will be disabled for SLAB and SLOB allocators?
SLAB is going to go away soon anyway and I guess not too many
people use SLOB.

> 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.
>

I remember discussing with you, but I can't remember the rational,
could you please explain it again.

> I know it's maybe too late, since some of you may be preparing

> for the Summit or LinixConf, but I think that we can go on
> discussing these on LinuxConf.
>

The LinuxConf and kernel summit is done now :-)

> The patches are applicable to the latest Morton's tree (that
> without the RSS controll) with the resource counters patch
> Andrew committed recently.
>

This is a bit confusing, it is applicable to 2.6.23-rc4-mm1?

> I've made some minimal testing for that and the similar code
> (without the containers interface but with the kmalloc
> accounting) is already in our 2.6.22 OpenVZ tree, so testing
> is going on.
>
> Thanks,
> Pavel

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

Subject: Re: [RFC][PATCH 0/3] Kernel memory accounting container (v2)
Posted by Pavel Emelianov on Thu, 13 Sep 2007 11:28:49 GMT
View Forum Message <> Reply to Message

Balbir Singh wrote:
> Pavel Emelyanov wrote:
>> 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.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.
>>
>
> Does this mean that the kernel memory container will have a dependency
> on SLUB and it will be disabled for SLAB and SLOB allocators?
> SLAB is going to go away soon anyway and I guess not too many
> people use SLOB.

Right now it is, but I can port it on booth - slab and slob
when slub is accepted.

>> 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.
>>
>
> I remember discussing with you, but I can't remember the rational,
> could you please explain it again.

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

>> I know it's maybe too late, since some of you may be preparing
>> for the Summit or LinixConf, but I think that we can go on
>> discussing these on LinuxConf.
>>
>
> The LinuxConf and kernel summit is done now :-)

Oops :) Copy-paste :(

>> The patches are applicable to the latest Morton's tree (that
>> without the RSS controll) with the resource counters patch
>> Andrew committed recently.
>>
>
> This is a bit confusing, it is applicable to 2.6.23-rc4-mm1?

Yup. Copy-paste again... sorry :(

>> I've made some minimal testing for that and the similar code
>> (without the containers interface but with the kmalloc
>> accounting) is already in our 2.6.22 OpenVZ tree, so testing
>> is going on.
>>
>> Thanks,
>> Pavel
>
>

Subject: Re: [RFC][PATCH 0/3] Kernel memory accounting container (v2)
Posted by Pavel Emelianov on Thu, 13 Sep 2007 11:33:07 GMT
View Forum Message <> Reply to Message

KAMEZAWA Hiroyuki wrote:
> On Thu, 13 Sep 2007 13:11:35 +0400
> Pavel Emelyanov <xemul@openvz.org> wrote:
>

>> 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
>>
> <snip>
>> 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.
>>
> Hmm, how can we know "How many kmem will we need ?" in precise per-object
> style ? Is this useful ?

You can start with unlimited container and check how many
kernel memory your applications use normally and set the limit
to 120% of this.

You may also set this to some reasonable value like 50% of normal
zone to protect your system from a fork bomb or similar.

This is the same question as "how many user pages will my
container consume". The answer is - find it out experimentally
or ask for someone who has already done so.

> Following kind of limitation of user friendly params is bad ?
>
> - # of file handles
> - # of tasks
> - # of sockets/ connections / packets
> - # of posix IPC related things
> - and other sources of DoS.

These are not enough and none of them are reasonable. E.g. the
struct vm_area_struct objects are allocated for many mmap() calls,
but how to find it out how many of them you will require.

However some controllers will be done as well.

> Thanks,
> -Kame

>
>
>

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

---

Subject: Re: [RFC][PATCH 0/3] Kernel memory accounting container (v2)
Posted by Christoph Lameter on Thu, 13 Sep 2007 18:36:42 GMT
View Forum Message <> Reply to Message

On Thu, 13 Sep 2007, Pavel Emelyanov wrote:

> 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.

Hmmmm... Okay I have seen multiple people who want to control slab allocations and track memory for various reasons. Would it be possible to come up with some hook that would allow a subscription to certain SLUB events? That way multiple subsystems may track and maybe disallow certain allocations in various contexts.

Subject: Re: [RFC][PATCH 0/3] Kernel memory accounting container (v2)
Posted by Pavel Emelianov on Fri, 14 Sep 2007 06:26:03 GMT
View Forum Message <> Reply to Message

Christoph Lameter wrote:
> On Thu, 13 Sep 2007, Pavel Emelyanov wrote:
>
>> 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.
>
> Hmmmm... Okay I have seen multiple people who want to control slab

> allocations and track memory for various reasons. Would it be possible to
> come up with some hook that would allow a subscription to certain SLUB
> events? That way multiple subsystems may track and maybe disallow certain
> allocations in various contexts.

Do you mean some more generic than just explicit call from slab_alloc, etc?
Ok, I will work on it.

Thanks,
Pavel

Subject: Re: [RFC][PATCH 0/3] Kernel memory accounting container (v2)
Posted by Christoph Lameter on Fri, 14 Sep 2007 17:30:26 GMT
View Forum Message <> Reply to Message

On Fri, 14 Sep 2007, Pavel Emelyanov wrote:

> > Hmmmm... Okay I have seen multiple people who want to control slab
> > allocations and track memory for various reasons. Would it be possible to
> > come up with some hook that would allow a subscription to certain SLUB
> > events? That way multiple subsystems may track and maybe disallow certain
> > allocations in various contexts.
>
> Do you mean some more generic than just explicit call from slab_alloc, etc?
> Ok, I will work on it.

Yes I guess an API in slub to register callbacks for these things. This
needs to include a way to control the granularity. Callbacks at the level
of slab allocations are likely not that performance critical. But some
uses may require control at the object level. There needs to be some way
to tell SLUB to disable the fast path for a particular allocated slab
because the API will police each individual allocation.

Subject: Re: [RFC][PATCH 0/3] Kernel memory accounting container (v2)
Posted by Pavel Emelianov on Mon, 17 Sep 2007 06:12:47 GMT
View Forum Message <> Reply to Message

Christoph Lameter wrote:
> On Fri, 14 Sep 2007, Pavel Emelyanov wrote:
>
>>> Hmmmm... Okay I have seen multiple people who want to control slab
>>> allocations and track memory for various reasons. Would it be possible to
>>> come up with some hook that would allow a subscription to certain SLUB
>>> events? That way multiple subsystems may track and maybe disallow certain
>>> allocations in various contexts.

>> Do you mean some more generic than just explicit call from slab_alloc, etc?
>> Ok, I will work on it.
>
> Yes I guess an API in slub to register callbacks for these things. This
> needs to include a way to control the granularity. Callbacks at the level
> of slab allocations are likely not that performance critical. But some
> uses may require control at the object level. There needs to be some way
> to tell SLUB to disable the fast path for a particular allocated slab
> because the API will police each individual allocation.
>

OK. I see. I will do my best to prepare the first version this week.

Thanks,
Pavel
