Subject: Re: [PATCH v5 04/14] kmem accounting basic infrastructure
Posted by David Rientjes on Wed, 17 Oct 2012 22:08:04 GMT
View Forum Message <> Reply to Message

On Tue, 16 Oct 2012, Glauber Costa wrote:

> This patch adds the basic infrastructure for the accounting of kernel
> memory. To control that, the following files are created:
>
> * memory.kmem.usage_in_bytes
> * memory.kmem.limit_in_bytes
> * memory.kmem.failcnt
> * memory.kmem.max_usage_in_bytes
>
> They have the same meaning of their user memory counterparts. They
> reflect the state of the "kmem" res_counter.
>
> Per cgroup kmem memory accounting is not enabled until a limit is set
> for the group. Once the limit is set the accounting cannot be disabled
> for that group. This means that after the patch is applied, no
> behavioral changes exists for whoever is still using memcg to control
> their memory usage, until memory.kmem.limit_in_bytes is set for the
> first time.
>
> We always account to both user and kernel resource_counters. This
> effectively means that an independent kernel limit is in place when the
> limit is set to a lower value than the user memory. A equal or higher
> value means that the user limit will always hit first, meaning that kmem
> is effectively unlimited.
>
> People who want to track kernel memory but not limit it, can set this
> limit to a very high number (like RESOURCE_MAX - 1page - that no one
> will ever hit, or equal to the user memory)
>
> [v4: make kmem files part of the main array;
>   do not allow limit to be set for non-empty cgroups]
> [v5: cosmetic changes]
>
> Signed-off-by: Glauber Costa <glommer@parallels.com>
> Acked-by: Kamezawa Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
> CC: Michal Hocko <mhocko@suse.cz>
> CC: Johannes Weiner <hannes@cmpxchg.org>
> CC: Tejun Heo <tj@kernel.org>
> ---
> mm/memcontrol.c | 116
++-
> 1 file changed, 115 insertions(+), 1 deletion(-)
>

> diff --git a/mm/memcontrol.c b/mm/memcontrol.c
> index 71d259e..30eafeb 100644
> --- a/mm/memcontrol.c
> +++ b/mm/memcontrol.c
> @@ -266,6 +266,10 @@ struct mem_cgroup {
> };
>
> /*
> + * the counter to account for kernel memory usage.
> + */
> + struct res_counter kmem;
> + /*
>   * Per cgroup active and inactive list, similar to the
>   * per zone LRU lists.
>   */
> @@ -280,6 +284,7 @@ struct mem_cgroup {
>   * Should the accounting and control be hierarchical, per subtree?
>   */
> bool use_hierarchy;
> + unsigned long kmem_accounted; /* See KMEM_ACCOUNTED_*, below */

I think this should be named kmem_account_flags or kmem_flags, otherwise
it appears that this is the actual account.

>
> bool oom_lock;
> atomic_t under_oom;
> @@ -332,6 +337,20 @@ struct mem_cgroup {
> #endif
> };
>
> +/* internal only representation about the status of kmem accounting. */
> +enum {
> + KMEM_ACCOUNTED_ACTIVE = 0, /* accounted by this cgroup itself */
> +};
> +
> +#define KMEM_ACCOUNTED_MASK (1 << KMEM_ACCOUNTED_ACTIVE)
> +
> +#ifdef CONFIG_MEMCG_KMEM

memcg->kmem_accounted isn't only defined for this configuration, so would
it be simpler to define this unconditionally?

> +static void memcg_kmem_set_active(struct mem_cgroup *memcg)

inline?

> +{

> + set_bit(KMEM_ACCOUNTED_ACTIVE, &memcg->kmem_accounted);
> +}
> +#endif
> +
> /* Stuffs for move charges at task migration. */
> /*
>   * Types of charges to be moved. "move_charge_at_immitgrate" is treated as a
> @@ -390,6 +409,7 @@ enum res_type {
>   _MEM,
>   _MEMSWAP,
>   _OOM_TYPE,
> + _KMEM,
> };
>
> #define MEMFILE_PRIVATE(x, val) ((x) << 16 | (val))
> @@ -1433,6 +1453,10 @@ done:
>   res_counter_read_u64(&memcg->memsw, RES_USAGE) >> 10,
>   res_counter_read_u64(&memcg->memsw, RES_LIMIT) >> 10,
>   res_counter_read_u64(&memcg->memsw, RES_FAILCNT));
> + printk(KERN_INFO "kmem: usage %llukB, limit %llukB, failcnt %llu\n",
> + res_counter_read_u64(&memcg->kmem, RES_USAGE) >> 10,
> + res_counter_read_u64(&memcg->kmem, RES_LIMIT) >> 10,
> + res_counter_read_u64(&memcg->kmem, RES_FAILCNT));
> }
>
> /*
> @@ -3940,6 +3964,9 @@ static ssize_t mem_cgroup_read(struct cgroup *cont, struct cftype
*cft,
>   else
>   val = res_counter_read_u64(&memcg->memsw, name);
>   break;
> + case _KMEM:
> + val = res_counter_read_u64(&memcg->kmem, name);
> + break;
>   default:
>   BUG();
>   }
> @@ -3947,6 +3974,57 @@ static ssize_t mem_cgroup_read(struct cgroup *cont, struct cftype
*cft,
>   len = scnprintf(str, sizeof(str), "%llu\n", (unsigned long long)val);
>   return simple_read_from_buffer(buf, nbytes, ppos, str, len);
> }
> +
> +static int memcg_update_kmem_limit(struct cgroup *cont, u64 val)
> +{
> + int ret = -EINVAL;
> +#ifdef CONFIG_MEMCG_KMEM
> + struct mem_cgroup *memcg = mem_cgroup_from_cont(cont);

```
> + /*
> +  * For simplicity, we won't allow this to be disabled.  It also can't
> +  * be changed if the cgroup has children already, or if tasks had
> +  * already joined.
> +  *
> +  * If tasks join before we set the limit, a person looking at
> +  * kmem.usage_in_bytes will have no way to determine when it took
> +  * place, which makes the value quite meaningless.
> +  *
> +  * After it first became limited, changes in the value of the limit are
> +  * of course permitted.
> +  *
> +  * Taking the cgroup_lock is really offensive, but it is so far the only
> +  * way to guarantee that no children will appear. There are plenty of
> +  * other offenders, and they should all go away. Fine grained locking
> +  * is probably the way to go here. When we are fully hierarchical, we
> +  * can also get rid of the use_hierarchy check.
```

Not sure it's so offensive, it's a pretty standard way of ensuring that cont->children doesn't get manipulated in a race.

```
> +  */
> + cgroup_lock();
> + mutex_lock(&set_limit_mutex);
> + if (!memcg->kmem_accounted && val != RESOURCE_MAX) {
> +  if (cgroup_task_count(cont) || (memcg->use_hierarchy &&
> +      !list_empty(&cont->children))) {
> +   ret = -EBUSY;
> +   goto out;
> +  }
> +  ret = res_counter_set_limit(&memcg->kmem, val);
> +  VM_BUG_ON(ret);
> +
> +  memcg_kmem_set_active(memcg);
> + } else
> +  ret = res_counter_set_limit(&memcg->kmem, val);
> +out:
> + mutex_unlock(&set_limit_mutex);
> + cgroup_unlock();
> +#endif
> + return ret;
> +}
> +
> +static void memcg_propagate_kmem(struct mem_cgroup *memcg,
> +     struct mem_cgroup *parent)
> +{
> + memcg->kmem_accounted = parent->kmem_accounted;
> +}
```

> +
> /*
>   * The user of this function is...
>   * RES_LIMIT.
> @@ -3978,8 +4056,12 @@ static int mem_cgroup_write(struct cgroup *cont, struct cftype *cft,
>   break;
>   if (type == _MEM)
>   ret = mem_cgroup_resize_limit(memcg, val);
> - else
> + else if (type == _MEMSWAP)
>   ret = mem_cgroup_resize_memsw_limit(memcg, val);
> + else if (type == _KMEM)
> +   ret = memcg_update_kmem_limit(cont, val);
> + else
> +   return -EINVAL;

I like how this is done in a maintainable way to ensure no other types can
inadvertently update the memsw limit as it was previously written. All
other returns of -EINVAL just cause the switch statement to break, though,
rather than return directly.

>   break;
>   case RES_SOFT_LIMIT:
>   ret = res_counter_memparse_write_strategy(buffer, &val);
> @@ -4045,12 +4127,16 @@ static int mem_cgroup_reset(struct cgroup *cont, unsigned int
event)
>   case RES_MAX_USAGE:
>   if (type == _MEM)
>   res_counter_reset_max(&memcg->res);
> + else if (type == _KMEM)
> +   res_counter_reset_max(&memcg->kmem);

Could this be written in the same way above, i.e. check _MEMSWAP to pass
memcg->memsw, _KMEM for memcg->kmem, etc?

>   else
>   res_counter_reset_max(&memcg->memsw);
>   break;
>   case RES_FAILCNT:
>   if (type == _MEM)
>   res_counter_reset_failcnt(&memcg->res);
> + else if (type == _KMEM)
> +   res_counter_reset_failcnt(&memcg->kmem);

Same.