Subject: Re: [RFC][PATCH][2/4] Add RSS accounting and control
Posted by Andrew Morton on Mon, 19 Feb 2007 08:58:28 GMT
View Forum Message <> Reply to Message

On Mon, 19 Feb 2007 12:20:34 +0530 Balbir Singh <balbir@in.ibm.com> wrote:

>
> This patch adds the basic accounting hooks to account for pages allocated
> into the RSS of a process. Accounting is maintained at two levels, in
> the mm_struct of each task and in the memory controller data structure
> associated with each node in the container.
>
> When the limit specified for the container is exceeded, the task is killed.
> RSS accounting is consistent with the current definition of RSS in the
> kernel. Shared pages are accounted into the RSS of each process as is
> done in the kernel currently. The code is flexible in that it can be easily
> modified to work with any definition of RSS.
>
> ..
>
> +static inline int memctlr_mm_init(struct mm_struct *mm)
> +{
> + return 0;
> +}

So it returns zero on success. OK.

> --- linux-2.6.20/kernel/fork.c~memctlr-acct 2007-02-18 22:55:50.000000000 +0530
> +++ linux-2.6.20-balbir/kernel/fork.c 2007-02-18 22:55:50.000000000 +0530
> @@ -50,6 +50,7 @@
> #include <linux/taskstats_kern.h>
> #include <linux/random.h>
> #include <linux/numtasks.h>
> +#include <linux/memctlr.h>
>
> #include <asm/pgtable.h>
> #include <asm/pgalloc.h>
> @@ -342,10 +343,15 @@ static struct mm_struct * mm_init(struct
>   mm->free_area_cache = TASK_UNMAPPED_BASE;
>   mm->cached_hole_size = ~0UL;
>
> + if (!memctlr_mm_init(mm))
> + goto err;
> +

But here we treat zero as an error?

>   if (likely(!mm_alloc_pgd(mm))) {

>   mm->def_flags = 0;
>   return mm;
>   }
> +
> +err:
>   free_mm(mm);
>   return NULL;
> }
>
> ...
>
> +int memctlr_mm_init(struct mm_struct *mm)
> +{
> + mm->counter = kmalloc(sizeof(struct res_counter), GFP_KERNEL);
> + if (!mm->counter)
> + return 0;
> + atomic_long_set(&mm->counter->usage, 0);
> + atomic_long_set(&mm->counter->limit, 0);
> + rwlock_init(&mm->container_lock);
> + return 1;
> +}

ah-ha, we have another Documentation/SubmitChecklist customer.

It would be more conventional to make this return -EFOO on error,
zero on success.

> +void memctlr_mm_free(struct mm_struct *mm)
> +{
> + kfree(mm->counter);
> +}
> +
> +static inline void memctlr_mm_assign_container_direct(struct mm_struct *mm,
> +   struct container *cont)
> +{
> + write_lock(&mm->container_lock);
> + mm->container = cont;
> + write_unlock(&mm->container_lock);
> +}

More weird locking here.

> +void memctlr_mm_assign_container(struct mm_struct *mm, struct task_struct *p)
> +{
> + struct container *cont = task_container(p, &memctlr_subsys);
> + struct memctlr *mem = memctlr_from_cont(cont);
> +
> + BUG_ON(!mem);

> + write_lock(&mm->container_lock);
> + mm->container = cont;
> + write_unlock(&mm->container_lock);
> +}

And here.

> +/*
> + * Update the rss usage counters for the mm_struct and the container it belongs
> + * to. We do not fail rss for pages shared during fork (see copy_one_pte()).
> + */
> +int memctlr_update_rss(struct mm_struct *mm, int count, bool check)
> +{
> + int ret = 1;
> + struct container *cont;
> + long usage, limit;
> + struct memctlr *mem;
> +
> + read_lock(&mm->container_lock);
> + cont = mm->container;
> + read_unlock(&mm->container_lock);
> +
> + if (!cont)
> + goto done;

And here. I mean, if there was a reason for taking the lock around that
read, then testing `cont' outside the lock just invalidated that reason.

> +static inline void memctlr_double_lock(struct memctlr *mem1,
> +   struct memctlr *mem2)
> +{
> + if (mem1 > mem2) {
> + spin_lock(&mem1->lock);
> + spin_lock(&mem2->lock);
> + } else {
> + spin_lock(&mem2->lock);
> + spin_lock(&mem1->lock);
> + }
> +}

Conventionally we take the lower-addressed lock first when doing this, not
the higher-addressed one.

> +static inline void memctlr_double_unlock(struct memctlr *mem1,
> +   struct memctlr *mem2)
> +{
> + if (mem1 > mem2) {
> + spin_unlock(&mem2->lock);

> + spin_unlock(&mem1->lock);
> + } else {
> + spin_unlock(&mem1->lock);
> + spin_unlock(&mem2->lock);
> + }
> +}
> +
> ...
>
>   retval = -ENOMEM;
> +
> + if (!memctlr_update_rss(mm, 1, MEMCTLR_CHECK_LIMIT))
> + goto out;
> +

Again, please use zero for success and -EFOO for error.

That way, you don't have to assume that the reason memctlr_update_rss()
failed was out-of-memory. Just propagate the error back.

>   flush_dcache_page(page);
>   pte = get_locked_pte(mm, addr, &ptl);
>   if (!pte)
> @@ -1580,6 +1587,9 @@ gotten:
>   cow_user_page(new_page, old_page, address, vma);
>   }
>
> + if (!memctlr_update_rss(mm, 1, MEMCTLR_CHECK_LIMIT))
> + goto oom;
> +
>   /*
>   * Re-check the pte - we dropped the lock
>   */
> @@ -1612,7 +1622,9 @@ gotten:
>   /* Free the old page.. */
>   new_page = old_page;
>   ret |= VM_FAULT_WRITE;
> - }
> + } else
> + memctlr_update_rss(mm, -1, MEMCTLR_DONT_CHECK_LIMIT);

This one doesn't get checked?

Why does MEMCTLR_DONT_CHECK_LIMIT exist?