Subject: Re: [RFC][PATCH 2/7] RSS controller core
Posted by xemul on Wed, 07 Mar 2007 07:25:25 GMT
View Forum Message <> Reply to Message

Balbir Singh wrote:
> Pavel Emelianov wrote:
>> This includes setup of RSS container within generic
>> process containers, all the declarations used in RSS
>> accounting, and core code responsible for accounting.
>>
>>
>> ---------- ----------
>>
>> diff -upr linux-2.6.20.orig/include/linux/rss_container.h
>> linux-2.6.20-0/include/linux/rss_container.h
>> --- linux-2.6.20.orig/include/linux/rss_container.h   2007-03-06
>> 13:39:17.000000000 +0300
>> +++ linux-2.6.20-0/include/linux/rss_container.h   2007-03-06
>> 13:33:28.000000000 +0300
>> @@ -0,0 +1,68 @@
>> +#ifndef __RSS_CONTAINER_H__
>> +#define __RSS_CONTAINER_H__
>> +/*
>> + * RSS container
>> + *
>> + * Copyright 2007 OpenVZ SWsoft Inc
>> + *
>> + * Author: Pavel Emelianov <xemul@openvz.org>
>> + *
>> + */
>> +
>> +struct page_container;
>> +struct rss_container;
>> +
>> +#ifdef CONFIG_RSS_CONTAINER
>> +int container_rss_prepare(struct page *, struct vm_area_struct *vma,
>> +   struct page_container **);
>> +
>> +void container_rss_add(struct page_container *);
>> +void container_rss_del(struct page_container *);
>> +void container_rss_release(struct page_container *);
>> +
>> +int mm_init_container(struct mm_struct *mm, struct task_struct *tsk);
>> +void mm_free_container(struct mm_struct *mm);
>> +
>> +unsigned long container_isolate_pages(unsigned long nr_to_scan,
>> +   struct rss_container *rss, struct list_head *dst,
>> +   int active, unsigned long *scanned);

```
>> +unsigned long container_nr_physpages(struct rss_container *rss);
>> +
>> +unsigned long container_try_to_free_pages(struct rss_container *);
>> +void container_out_of_memory(struct rss_container *);
>> +
>> +void container_rss_init_early(void);
>> +#else
>> +static inline int container_rss_prepare(struct page *pg,
>> +        struct vm_area_struct *vma, struct page_container **pc)
>> +{
>> +    *pc = NULL; /* to make gcc happy */
>> +    return 0;
>> +}
>> +
>> +static inline void container_rss_add(struct page_container *pc)
>> +{
>> +}
>> +
>> +static inline void container_rss_del(struct page_container *pc)
>> +{
>> +}
>> +
>> +static inline void container_rss_release(struct page_container *pc)
>> +{
>> +}
>> +
>> +static inline int mm_init_container(struct mm_struct *mm, struct
>> task_struct *t)
>> +{
>> +    return 0;
>> +}
>> +
>> +static inline void mm_free_container(struct mm_struct *mm)
>> +{
>> +}
>> +
>> +static inline void container_rss_init_early(void)
>> +{
>> +}
>> +#endif
>> +#endif
>> diff -upr linux-2.6.20.orig/init/Kconfig linux-2.6.20-0/init/Kconfig
>> --- linux-2.6.20.orig/init/Kconfig    2007-03-06 13:33:28.000000000 +0300
>> +++ linux-2.6.20-0/init/Kconfig    2007-03-06 13:33:28.000000000 +0300
>> @@ -265,6 +265,13 @@ config CPUSETS
>>      bool
>>      select CONTAINERS
>>
```

>> +config RSS_CONTAINER
>> +   bool "RSS accounting container"
>> +   select RESOURCE_COUNTERS
>> +   help
>> +   Provides a simple Resource Controller for monitoring and
>> +   controlling the total Resident Set Size of the tasks in a
>> container
>> +
>
> The wording looks very familiar :-). It would be useful to add
> "The reclaim logic is now container aware, when the container goes
> overlimit
> the page reclaimer reclaims pages belonging to this container. If we are
> unable to reclaim enough pages to satisfy the request, the process is
> killed with an out of memory warning"

OK. Thanks.

>
>> config SYSFS_DEPRECATED
>>   bool "Create deprecated sysfs files"
>>   default y
>> diff -upr linux-2.6.20.orig/mm/Makefile linux-2.6.20-0/mm/Makefile
>> --- linux-2.6.20.orig/mm/Makefile   2007-02-04 21:44:54.000000000 +0300
>> +++ linux-2.6.20-0/mm/Makefile   2007-03-06 13:33:28.000000000 +0300
>> @@ -29,3 +29,5 @@ obj-$(CONFIG_MEMORY_HOTPLUG) += memory_h
>> obj-$(CONFIG_FS_XIP) += filemap_xip.o
>> obj-$(CONFIG_MIGRATION) += migrate.o
>> obj-$(CONFIG_SMP) += allocpercpu.o
>> +
>> +obj-$(CONFIG_RSS_CONTAINER) += rss_container.o
>> diff -upr linux-2.6.20.orig/mm/rss_container.c
>> linux-2.6.20-0/mm/rss_container.c
>> --- linux-2.6.20.orig/mm/rss_container.c   2007-03-06
>> 13:39:17.000000000 +0300
>> +++ linux-2.6.20-0/mm/rss_container.c   2007-03-06 13:33:28.000000000
>> +0300
>> @@ -0,0 +1,307 @@
>> +/*
>> + * RSS accounting container
>> + *
>> + * Copyright 2007 OpenVZ SWsoft Inc
>> + *
>> + * Author: Pavel Emelianov <xemul@openvz.org>
>> + *
>> + */
>> +
>> +#include <linux/list.h>

```
>> +#include <linux/sched.h>
>> +#include <linux/mm.h>
>> +#include <linux/res_counter.h>
>> +#include <linux/rss_container.h>
>> +
>> +static struct container_subsys rss_subsys;
>> +
>> +struct rss_container {
>> +    struct res_counter res;
>> +    struct list_head page_list;
>> +    struct container_subsys_state css;
>> +};
>> +
>> +struct page_container {
>> +    struct page *page;
>> +    struct rss_container *cnt;
>> +    struct list_head list;
>> +};
>> +
>
> Yes, this is what I was planning to get to -- a per container LRU list.
> But you have just one list, don't you need active and inactive lists?
> When the global LRU is manipulated, shouldn't this list be updated as
> well, so that reclaim will pick the best pages.
>
>> +static inline struct rss_container *rss_from_cont(struct container *cnt)
>> +{
>> +    return container_of(container_subsys_state(cnt, &rss_subsys),
>> +            struct rss_container, css);
>> +}
>> +
>> +int mm_init_container(struct mm_struct *mm, struct task_struct *tsk)
>> +{
>> +    struct rss_container *cnt;
>> +
>> +    cnt = rss_from_cont(task_container(tsk, &rss_subsys));
>> +    if (css_get(&cnt->css))
>> +        return -EBUSY;
>> +
>> +    mm->rss_container = cnt;
>> +    return 0;
>> +}
>> +
>> +void mm_free_container(struct mm_struct *mm)
>> +{
>> +    css_put(&mm->rss_container->css);
>> +}
>> +
```

```
>> +int container_rss_prepare(struct page *page, struct vm_area_struct *vma,
>> +        struct page_container **ppc)
>> +{
>> +    struct rss_container *rss;
>> +    struct page_container *pc;
>> +
>> +    rcu_read_lock();
>> +    rss = rcu_dereference(vma->vm_mm->rss_container);
>> +    css_get_current(&rss->css);
>> +    rcu_read_unlock();
>> +
>> +    pc = kmalloc(sizeof(struct page_container), GFP_KERNEL);
>> +    if (pc == NULL)
>> +        goto out_nomem;
>> +
>> +    while (res_counter_charge(&rss->res, 1)) {
>> +        if (container_try_to_free_pages(rss))
>> +            continue;
>> +
>
> The return codes of the functions is a bit confusing, ideally
> container_try_to_free_pages() should return 0 on success. Also
```

This returns exactly what try_to_free_pages() does.

> res_counter_charge() has a WARN_ON(1) if the limit is exceeded.

Nope - res_counter_uncharge() has - this is an absolutely
sane check that we haven't over-uncharged resources.

> The system administrator can figure out the details from failcnt,
> I suspect when the container is running close to it's limit,
> dmesg will have too many WARNING messages.
>
> How much memory do you try to reclaim in container_try_to_free_pages()?

At least one page. This is enough to make one page charge.
That's the difference from general try_to_free_pages() that
returns success if it freed swap_cluster_max pages at least.

> With my patches, I was planning to export this knob to userspace with
> a default value. This will help the administrator decide how much
> of the working set/container LRU should be freed on reaching the limit.
> I cannot find the definition of container_try_to_free_pages() in
> this patch.

This is in patch #5.
Sorry for such a bad split - I'll make it cleaner next time :)

```
>
>
>> +        container_out_of_memory(rss);
>> +        if (test_thread_flag(TIF_MEMDIE))
>> +            goto out_charge;
>> +    }
>> +
>> +    pc->page = page;
>> +    pc->cnt = rss;
>> +    *ppc = pc;
>> +    return 0;
>> +
>> +out_charge:
>> +    kfree(pc);
>> +out_nomem:
>> +    css_put(&rss->css);
>> +    return -ENOMEM;
>> +}
>> +
>> +void container_rss_release(struct page_container *pc)
>> +{
>> +    struct rss_container *rss;
>> +
>> +    rss = pc->cnt;
>> +    res_counter_uncharge(&rss->res, 1);
>> +    css_put(&rss->css);
>> +    kfree(pc);
>> +}
>> +
>> +void container_rss_add(struct page_container *pc)
>> +{
>> +    struct page *pg;
>> +    struct rss_container *rss;
>> +
>> +    pg = pc->page;
>> +    rss = pc->cnt;
>> +
>> +    spin_lock(&rss->res.lock);
>> +    list_add(&pc->list, &rss->page_list);
>
> This is not good, it won't give us LRU behaviour which is
> useful for determining which pages to free.
```

Why not - recently used pages are in the head of the list.
Active/incative state of the page is determined from it's flags.

The idea of this list is to decrease the number of pages scanned

during reclamation. LRU-ness is checked from global page state.

```
>> +    spin_unlock(&rss->res.lock);
>> +
>> +    page_container(pg) = pc;
>> +}
>> +
>> +void container_rss_del(struct page_container *pc)
>> +{
>> +    struct page *page;
>> +    struct rss_container *rss;
>> +
>> +    page = pc->page;
>> +    rss = pc->cnt;
>> +
>> +    spin_lock(&rss->res.lock);
>> +    list_del(&pc->list);
>> +    res_counter_uncharge_locked(&rss->res, 1);
>> +    spin_unlock(&rss->res.lock);
>> +
>> +    css_put(&rss->css);
>> +    kfree(pc);
>> +}
>> +
>> +unsigned long container_isolate_pages(unsigned long nr_to_scan,
>> +        struct rss_container *rss, struct list_head *dst,
>> +        int active, unsigned long *scanned)
>> +{
>> +    unsigned long nr_taken = 0;
>> +    struct page *page;
>> +    struct page_container *pc;
>> +    unsigned long scan;
>> +    struct list_head *src;
>> +    LIST_HEAD(pc_list);
>> +    struct zone *z;
>> +
>> +    spin_lock_irq(&rss->res.lock);
>> +    src = &rss->page_list;
>> +
>
> Which part of the working set are we pushing out, this looks like
> we are using FIFO to determine which pages to reclaim. This needs
> to be FIXED.
```

This algo works exactly like general try_to_free_pages() does.
It scans for active pages first, then for inactive shrinking them.

```
>> +    for (scan = 0; scan < nr_to_scan && !list_empty(src); scan++) {
```

```
>> +       pc = list_entry(src->prev, struct page_container, list);
>> +       page = pc->page;
>> +       z = page_zone(page);
>> +
>> +       list_move(&pc->list, &pc_list);
>> +
>> +       spin_lock(&z->lru_lock);
>> +       if (PageLRU(page)) {
>> +           if ((active && PageActive(page)) ||
>> +                   (!active && !PageActive(page))) {
>> +               if (likely(get_page_unless_zero(page))) {
>> +                   ClearPageLRU(page);
>> +                   nr_taken++;
>> +                   list_move(&page->lru, dst);
>> +               }
>> +           }
>> +       }
>> +       spin_unlock(&z->lru_lock);
>> +   }
>> +
>> +   list_splice(&pc_list, src);
>
> This would lead to LRU churning, I would recommend using list_splice_tail()
> instead. Since this code has a lot in common with isolate_lru_pages, it
> would be nice to reuse the code in vmscan.c
```

I'll look at it.

```
> NOTE: Code duplication is a back door for subtle bugs and solving the same
> issue twice :-)
>
>> +   spin_unlock_irq(&rss->res.lock);
>> +
>> +   *scanned = scan;
>> +   return nr_taken;
>> +}
>> +
>> +unsigned long container_nr_physpages(struct rss_container *rss)
>> +{
>> +   return rss->res.usage;
>> +}
>> +
>> +static void rss_move_task(struct container_subsys *ss,
>> +       struct container *cont,
>> +       struct container *old_cont,
>> +       struct task_struct *p)
>> +{
>> +   struct mm_struct *mm;
```

```
>> +	struct rss_container *rss, *old_rss;
>> +
>> +	mm = get_task_mm(p);
>> +	if (mm == NULL)
>> +		goto out;
>> +
>> +	rss = rss_from_cont(cont);
>> +	old_rss = rss_from_cont(old_cont);
>> +	if (old_rss != mm->rss_container)
>> +		goto out_put;
>> +
>> +	css_get_current(&rss->css);
>> +	rcu_assign_pointer(mm->rss_container, rss);
>> +	css_put(&old_rss->css);
>> +
>
> I see that the charges are not migrated. Is that good?
```

This is what we came to long time ago - no resource migration.

```
> If a user could find a way of migrating his/her task from
> one container to another, it could create an issue with
> the user's task taking up a big chunk of the RSS limit.
>
> Can we migrate any task or just the thread group leader.
> In my patches, I allowed migration of just the thread
> group leader. Imagine if you have several threads, no
> matter which container they belong to, their mm gets
> charged (usage will not show up in the container's usage).
> This could confuse the system administrator.
```

Anyway - page migration may be done later with a
separate patch.

```
>> +out_put:
>> +	mmput(mm);
>> +out:
>> +	return;
>> +}
>> +
>> +static int rss_create(struct container_subsys *ss, struct container
>> *cont)
>> +{
>> +	struct rss_container *rss;
>> +
>> +	rss = kzalloc(sizeof(struct rss_container), GFP_KERNEL);
>> +	if (rss == NULL)
>> +		return -ENOMEM;
```

```
>> +
>> +    res_counter_init(&rss->res);
>> +    INIT_LIST_HEAD(&rss->page_list);
>> +    cont->subsys[rss_subsys.subsys_id] = &rss->css;
>> +    return 0;
>> +}
>> +
>> +static void rss_destroy(struct container_subsys *ss,
>> +        struct container *cont)
>> +{
>> +    kfree(rss_from_cont(cont));
>> +}
>> +
>> +
>> +static ssize_t rss_read(struct container *cont, struct cftype *cft,
>> +        struct file *file, char __user *userbuf,
>> +        size_t nbytes, loff_t *ppos)
>> +{
>> +    return res_counter_read(&rss_from_cont(cont)->res, cft->private,
>> +            userbuf, nbytes, ppos);
>> +}
>> +
>> +static ssize_t rss_write(struct container *cont, struct cftype *cft,
>> +        struct file *file, const char __user *userbuf,
>> +        size_t nbytes, loff_t *ppos)
>> +{
>> +    return res_counter_write(&rss_from_cont(cont)->res, cft->private,
>> +            userbuf, nbytes, ppos);
>> +}
>> +
>> +
>> +static struct cftype rss_usage = {
>> +    .name = "rss_usage",
>> +    .private = RES_USAGE,
>> +    .read = rss_read,
>> +};
>> +
>> +static struct cftype rss_limit = {
>> +    .name = "rss_limit",
>> +    .private = RES_LIMIT,
>> +    .read = rss_read,
>> +    .write = rss_write,
>> +};
>> +
>> +static struct cftype rss_failcnt = {
>> +    .name = "rss_failcnt",
>> +    .private = RES_FAILCNT,
>> +    .read = rss_read,
```

```
>> +};
>> +
>> +static int rss_populate(struct container_subsys *ss,
>> +        struct container *cont)
>> +{
>> +    int rc;
>> +
>> +    if ((rc = container_add_file(cont, &rss_usage)) < 0)
>> +        return rc;
>> +    if ((rc = container_add_file(cont, &rss_failcnt)) < 0)
>> +        return rc;
>> +    if ((rc = container_add_file(cont, &rss_limit)) < 0)
>> +        return rc;
>> +
>> +    return 0;
>> +}
>> +
>> +static struct rss_container init_rss_container;
>> +
>> +static __init int rss_create_early(struct container_subsys *ss,
>> +        struct container *cont)
>> +{
>> +    struct rss_container *rss;
>> +
>> +    rss = &init_rss_container;
>> +    res_counter_init(&rss->res);
>> +    INIT_LIST_HEAD(&rss->page_list);
>> +    cont->subsys[rss_subsys.subsys_id] = &rss->css;
>> +    ss->create = rss_create;
>> +    return 0;
>> +}
>> +
>> +static struct container_subsys rss_subsys = {
>> +    .name = "rss",
>> +    .create = rss_create_early,
>> +    .destroy = rss_destroy,
>> +    .populate = rss_populate,
>> +    .attach = rss_move_task,
>> +};
>> +
>> +void __init container_rss_init_early(void)
>> +{
>> +    container_register_subsys(&rss_subsys);
>> +    init_mm.rss_container = rss_from_cont(
>> +            task_container(&init_task, &rss_subsys));
>> +    css_get_current(&init_mm.rss_container->css);
>> +}
>>
```

>
>