Subject: [PATCH 5/8] RSS container core
Posted by Pavel Emelianov on Mon, 04 Jun 2007 13:35:07 GMT
View Forum Message <> Reply to Message

The core routines for tracking the page ownership, RSS subsystem
registration in the containers and the definition of the
rss_container struct as container subsystem combined with the
resource counter structure.

To make the whole set look more consistent the calls to the
reclamation code and oom killer are removed from this patch,
so if someone has the kernel snapshot stopped at this patch
and calls the accounting routines and the limit will be hit,
the current task will be killed immediately.

Includes fixes from Balbir Singh <balbir@in.ibm.com>

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

```
--- ./include/linux/container_subsys.h.rsscore 2007-06-04 12:05:30.000000000 +0400
+++ ./include/linux/container_subsys.h 2007-06-04 12:06:27.000000000 +0400
@@ -11,6 +11,10 @@
 SUBSYS(cpuacct)
 #endif

+#ifdef CONFIG_RSS_CONTAINER
+SUBSYS(rss)
+#endif
+
 /* */

 #ifdef CONFIG_CPUSETS
--- ./include/linux/rss_container.h.rsscore 2007-06-04 12:06:06.000000000 +0400
+++ ./include/linux/rss_container.h 2007-06-04 12:06:27.000000000 +0400
@@ -9,6 +9,93 @@
  *
  */

+struct page_container;
+struct rss_container;
+
+#ifdef CONFIG_RSS_CONTAINER
+/*
+ * This is how the RSS accounting works.
+ *
+ * Abstract:
```

```
+ *  Each mapped page has an owning container and is linked into its LRU lists
+ *  just like in the global LRU ones. The owner of the page is the container
+ *  that touched the page first. As long as the page stays mapped it holds
+ *  the container, is accounted into its usage and lives in its LRU list.
+ *  When page is unmapped for the last time it releases the container. The
+ *  RSS usage is exactly the number of pages in its booth LRU lists. When
+ *  this usage exceeds the limit set some pages are reclaimed from the owning
+ *  container. In case no reclamation possible the OOM killer starts thinning
+ *  out the container.
+ *
+ *
+ * The details:
+ *  The mapping-to-user process is splitted into 3 stages:
+ *  1. The page allocation and charging. This stage is might-sleep one and
+ *     in case the container has run out of resources the reclamation code
+ *     starts.
+ *
+ *    container_rss_prepare() fultion prepares the page container to add it
+ *    to the rss container and charges the resource counter.
+ *
+ *  2. The page tables tuning. This stage is atomic (as it is always done under
+ *     the ptl lock) and handles the race between multiple mappers.
+ *
+ *    container_rss_add() assigns the container to the page and adds it to
+ *    the container's LRU list.
+ *
+ *  3. The release/error path. In case of any error after prepare or race
+ *     with another toucher the page container must be released.
+ *
+ *    container_rss_release() frees the preallocated container and uncharges
+ *    the now-unneded amount from the counter
+ *
+ *  The unmapping process is simpe:
+ *     When page is unmapped from the last address space it releases the
+ *     container and is removed from its LRU lists. This is importaint that
+ *     the page stays in the global LRU list till it is completely freed
+ *
+ *    container_rss_del() removes the container from the page
+ *
+ *
+ * The "mapped for the first time" and "unmapped from the last user" conditions
+ * are checkin in rmap calls.
+ *   - If mapcount changes form 0 to 1 the page is first touched and is added
+ *     to the container;
+ *   - If mapcount changes from 1 to 0 the page is unmapped for the last time
+ *     and is removed from the container.
+ *
+ * See comments in mm/rmap.c for more details.
```

```
+ *
+ */
+
+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 *);
+
+void mm_init_container(struct mm_struct *mm, struct task_struct *tsk);
+void mm_free_container(struct mm_struct *mm);
+#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 void mm_init_container(struct mm_struct *mm, struct task_struct *t)
 {
 }
@@ -17,3 +103,4 @@ static inline void mm_free_container(str
 {
 }
 #endif
+#endif
--- ./init/Kconfig.rsscore 2007-06-04 12:05:48.000000000 +0400
+++ ./init/Kconfig 2007-06-04 12:06:27.000000000 +0400
@@ -332,6 +332,17 @@ config RESOURCE_COUNTERS
  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 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.
+
 config SYSFS_DEPRECATED
  bool "Create deprecated sysfs files"
  default y
--- ./mm/Makefile.rsscore 2007-06-04 12:05:26.000000000 +0400
+++ ./mm/Makefile 2007-06-04 12:06:27.000000000 +0400
@@ -30,4 +30,5 @@ obj-$(CONFIG_FS_XIP) += filemap_xip.o
 obj-$(CONFIG_MIGRATION) += migrate.o
 obj-$(CONFIG_SMP) += allocpercpu.o
 obj-$(CONFIG_QUICKLIST) += quicklist.o
+obj-$(CONFIG_RSS_CONTAINER) += rss_container.o

--- /dev/null 2007-04-11 18:31:29.344197500 +0400
+++ ./mm/rss_container.c 2007-06-04 12:06:27.000000000 +0400
@@ -0,0 +1,323 @@
+/*
+ * RSS accounting container
+ *
+ * Copyright 2007 OpenVZ SWsoft Inc
+ *
+ * Author: Pavel Emelianov <xemul@openvz.org>
+ * Fixes from: Balbir Singh <balbir@in.ibm.com>
+ *
+ */
+
+#include <linux/list.h>
+#include <linux/sched.h>
+#include <linux/mm.h>
+#include <linux/swap.h>
+#include <linux/res_counter.h>
+#include <linux/rss_container.h>
+
+/*
+ * the rss container description
+ */
+
+struct rss_container {
+ /*
+  * the counter to account for RSS
+  */
+ struct res_counter res;
```

```
+ /*
+  * the lists of pages within the container.
+  * actually these lists store the page containers (see below), not
+  * pages. they live just like the global LRU lists do. i.e. the
+  * page containers migrate from one list to another as soon as the
+  * according page does. see container_rss_move_lists and comment
+  * in include/linux/rss_container.h for more details
+  *
+  * these lists are protected with res_counter's lock and this lock
+  * may be taken under zone->lru_lock
+  */
+ struct list_head inactive_list;
+ struct list_head active_list;
+ /*
+  * the number of successful reclaims from the container. reclamation
+  * happens each time the counter reports that no resource available,
+  * so low value in comparison to the failcnt will indicate that the
+  * container is experiencing difficulties during its lifetime and is
+  * probably misconfigured
+  */
+ atomic_t rss_reclaimed;
+
+ struct container_subsys_state css;
+};
+
+/*
+ * page container ties together a page and a container making a one-to-one
+ * relations between them. see comment in inlude/linux/rss_container.h on
+ * how this relation is set
+ */
+
+struct page_container {
+ struct page *page;
+ struct rss_container *cnt;
+ struct list_head list; /* this is the element of (int)active_list of
+     * the container.
+     */
+};
+
+static inline struct rss_container *rss_from_cont(struct container *cnt)
+{
+ return container_of(container_subsys_state(cnt, rss_subsys_id),
+    struct rss_container, css);
+}
+
+/*
+ * mm_struct represents the page consumer, so we store the container pointer
+ * on them as well and use it in accounting
```

```
+ */
+
+void mm_init_container(struct mm_struct *mm, struct task_struct *tsk)
+{
+ struct rss_container *cnt;
+
+ cnt = rss_from_cont(task_container(tsk, rss_subsys_id));
+ css_get(&cnt->css);
+ mm->rss_container = cnt;
+}
+
+void mm_free_container(struct mm_struct *mm)
+{
+ css_put(&mm->rss_container->css);
+}
+
+/*
+ * ownership tracking.
+ * I bet you have already read the comment in include/linux/rss_container.h :)
+ */
+
+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(&rss->css);
+ rcu_read_unlock();
+
+ pc = kmalloc(sizeof(struct page_container), GFP_KERNEL);
+ if (pc == NULL)
+  goto out_nomem;
+
+ if (res_counter_charge(&rss->res, 1))
+  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_irq(&rss->res.lock);
+ list_add(&pc->list, &rss->active_list);
+ spin_unlock_irq(&rss->res.lock);
+
+ set_page_container(pg, pc);
+}
+
+void container_rss_del(struct page_container *pc)
+{
+ struct rss_container *rss;
+
+ rss = pc->cnt;
+ spin_lock_irq(&rss->res.lock);
+ list_del(&pc->list);
+ res_counter_uncharge_locked(&rss->res, 1);
+ spin_unlock_irq(&rss->res.lock);
+
+ css_put(&rss->css);
+ kfree(pc);
+}
+
+/*
+ * interaction with the containers subsystem
+ */
+
+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);
+
+ /*
+  * tasks may share one mm_struct with each other. on the other hand
+  * tasks may belong to different containers. keep mm_structs tied
+  * to the according task and migrate them together
+  */
+
+ if (old_rss != mm->rss_container)
+  goto out_put;
+
+ css_get(&rss->css);
+ rcu_assign_pointer(mm->rss_container, rss);
+ css_put(&old_rss->css);
+
+out_put:
+ mmput(mm);
+out:
+ return;
+}
+
+static struct rss_container init_rss_container;
+
+static inline void rss_container_attach(struct rss_container *rss,
+  struct container *cont)
+{
+ cont->subsys[rss_subsys_id] = &rss->css;
+ rss->css.container = cont;
+}
+
+static int rss_create(struct container_subsys *ss, struct container *cont)
+{
+ struct rss_container *rss;
+
+ if (unlikely(cont->parent == NULL)) {
+  rss = &init_rss_container;
+  css_get(&rss->css);
```

```
+  init_mm.rss_container = rss;
+ } else
+  rss = kzalloc(sizeof(struct rss_container), GFP_KERNEL);
+
+ if (rss == NULL)
+  return -ENOMEM;
+
+ res_counter_init(&rss->res);
+ INIT_LIST_HEAD(&rss->inactive_list);
+ INIT_LIST_HEAD(&rss->active_list);
+ rss_container_attach(rss, cont);
+ 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 ssize_t rss_read_reclaimed(struct container *cont, struct cftype *cft,
+  struct file *file, char __user *userbuf,
+  size_t nbytes, loff_t *ppos)
+{
+ char buf[64], *s;
+
+ s = buf;
+ s += sprintf(s, "%d\n",
+   atomic_read(&rss_from_cont(cont)->rss_reclaimed));
+ return simple_read_from_buffer((void __user *)userbuf, nbytes,
+   ppos, buf, s - buf);
```

```
+}
+
+/*
+ * container files to export RSS container's counter state to userspace
+ * one file for each field
+ */
+
+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 struct cftype rss_reclaimed = {
+ .name = "rss_reclaimed",
+ .read = rss_read_reclaimed,
+};
+
+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;
+ if ((rc = container_add_file(cont, &rss_reclaimed)) < 0)
+  return rc;
+
+ return 0;
+}
+
```

```
+struct container_subsys rss_subsys = {
+ .name = "rss",
+ .subsys_id = rss_subsys_id,
+ .create = rss_create,
+ .destroy = rss_destroy,
+ .populate = rss_populate,
+ .attach = rss_move_task,
+ .early_init = 1,
+};
```