Subject: [PATCH 0/8] RSS controller based on process containers (v3)
Posted by Pavel Emelianov on Wed, 30 May 2007 15:24:41 GMT
View Forum Message <> Reply to Message

Adds RSS accounting and control within a container.

Ported above Paul's containers V10 with fixes received from Balbir.

RSS container includes the per-container RSS accounting
and reclamation, and out-of-memory killer. The container
behaves like a standalone machine - when it runs out of
resources, it tries to reclaim some pages, and if it doesn't
succeed, kills some task which mm_struct belongs to the
container in question.

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

Subject: [PATCH 4/8] RSS container core
Posted by Pavel Emelianov on Wed, 30 May 2007 15:27:53 GMT
View Forum Message <> Reply to Message

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

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

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

---

diff -upr linux-2.6.22-rc2-mm1.orig/include/linux/container_subsys.h
linux-2.6.22-rc2-mm1-0/include/linux/container_subsys.h
--- linux-2.6.22-rc2-mm1.orig/include/linux/container_subsys.h 2007-05-30 16:13:08.000000000
+0400
+++ linux-2.6.22-rc2-mm1-0/include/linux/container_subsys.h 2007-05-30 16:13:09.000000000
+0400
@@ -11,6 +11,10 @@
 SUBSYS(cpuacct)
 #endif

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

```
 #ifdef CONFIG_CPUSETS
diff -upr linux-2.6.22-rc2-mm1.orig/include/linux/rss_container.h
linux-2.6.22-rc2-mm1-0/include/linux/rss_container.h
--- linux-2.6.22-rc2-mm1.orig/include/linux/rss_container.h 2007-05-30 16:16:58.000000000
+0400
+++ linux-2.6.22-rc2-mm1-0/include/linux/rss_container.h 2007-05-30 16:13:09.000000000 +0400
@@ -0,0 +1,64 @@
+#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 *);
+
+void mm_init_container(struct mm_struct *mm, struct task_struct *tsk);
+void mm_free_container(struct mm_struct *mm);
+
+static inline void init_page_container(struct page *page)
+{
+ page_container(page) = NULL;
+}
+
+#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)
+{
+}
+
+static inline void mm_free_container(struct mm_struct *mm)
+{
+}
+
+static inline void init_page_container(struct page *page)
+{
+}
+
+#endif
+#endif
diff -upr linux-2.6.22-rc2-mm1.orig/init/Kconfig linux-2.6.22-rc2-mm1-0/init/Kconfig
--- linux-2.6.22-rc2-mm1.orig/init/Kconfig 2007-05-30 16:13:08.000000000 +0400
+++ linux-2.6.22-rc2-mm1-0/init/Kconfig 2007-05-30 16:13:09.000000000 +0400
@@ -328,6 +328,17 @@ 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 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
diff -upr linux-2.6.22-rc2-mm1.orig/mm/Makefile linux-2.6.22-rc2-mm1-0/mm/Makefile
--- linux-2.6.22-rc2-mm1.orig/mm/Makefile 2007-05-30 12:32:36.000000000 +0400
+++ linux-2.6.22-rc2-mm1-0/mm/Makefile 2007-05-30 16:13:09.000000000 +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

diff -upr linux-2.6.22-rc2-mm1.orig/mm/rss_container.c
linux-2.6.22-rc2-mm1-0/mm/rss_container.c
--- linux-2.6.22-rc2-mm1.orig/mm/rss_container.c 2007-05-30 16:16:58.000000000 +0400
+++ linux-2.6.22-rc2-mm1-0/mm/rss_container.c 2007-05-30 16:13:09.000000000 +0400
@@ -0,0 +1,271 @@
+/*
+ * 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/swap.h>
+#include <linux/res_counter.h>
+#include <linux/rss_container.h>
+
+struct rss_container {
+ struct res_counter res;
+ struct list_head inactive_list;
+ struct list_head active_list;
+ atomic_t rss_reclaimed;
+ struct container_subsys_state css;
+};
+
+struct page_container {
+ struct page *page;
+ struct rss_container *cnt;
+ struct list_head list;
+};
+
+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);
+}
+
+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);
+}
+
+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;
+
+ while (res_counter_charge(&rss->res, 1)) {
+  if (try_to_free_pages_in_container(rss)) {
+   atomic_inc(&rss->rss_reclaimed);
+   continue;
+  }
+
+  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_irq(&rss->res.lock);
+ list_add(&pc->list, &rss->active_list);
+ spin_unlock_irq(&rss->res.lock);
+
+ 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);
+}
+
+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(&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);
+}
+
+
+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,
+};
```

---

Subject: [PATCH 5/8] RSS accounting hooks over the code
Posted by Pavel Emelianov on Wed, 30 May 2007 15:29:45 GMT
View Forum Message <> Reply to Message

As described above, pages are charged to their first touchers.
The first toucher is determined using pages' _mapcount
manipulations in rmap calls.

Page is charged in two stages:
1. preparation, in which the resource availability is checked.
   This stage may lead to page reclamation, thus it is performed
   in a "might-sleep" places;
2. the container assignment to page. This is done in an atomic
   code that handles races between multiple touchers.

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

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

---

```
diff -upr linux-2.6.22-rc2-mm1.orig/fs/exec.c linux-2.6.22-rc2-mm1-0/fs/exec.c
--- linux-2.6.22-rc2-mm1.orig/fs/exec.c 2007-05-30 12:32:34.000000000 +0400
+++ linux-2.6.22-rc2-mm1-0/fs/exec.c 2007-05-30 16:13:09.000000000 +0400
@@ -59,6 +59,8 @@
 #include <linux/kmod.h>
 #endif

+#include <linux/rss_container.h>
+
 int core_uses_pid;
 char core_pattern[CORENAME_MAX_SIZE] = "core";
 int suid_dumpable = 0;
@@ -314,27 +316,34 @@ void install_arg_page(struct vm_area_str
  struct mm_struct *mm = vma->vm_mm;
  pte_t * pte;
  spinlock_t *ptl;
+ struct page_container *pcont;

  if (unlikely(anon_vma_prepare(vma)))
   goto out;

+ if (container_rss_prepare(page, vma, &pcont))
+  goto out;
+
  flush_dcache_page(page);
  pte = get_locked_pte(mm, address, &ptl);
  if (!pte)
-  goto out;
+  goto out_release;
  if (!pte_none(*pte)) {
   pte_unmap_unlock(pte, ptl);
```

```
-  goto out;
+  goto out_release;
   }
   inc_mm_counter(mm, anon_rss);
   lru_cache_add_active(page);
   set_pte_at(mm, address, pte, pte_mkdirty(pte_mkwrite(mk_pte(
       page, vma->vm_page_prot))));
- page_add_new_anon_rmap(page, vma, address);
+ page_add_new_anon_rmap(page, vma, address, pcont);
   pte_unmap_unlock(pte, ptl);

   /* no need for flush_tlb */
   return;
+
+out_release:
+ container_rss_release(pcont);
  out:
   __free_page(page);
   force_sig(SIGKILL, current);
diff -upr linux-2.6.22-rc2-mm1.orig/include/linux/rmap.h
linux-2.6.22-rc2-mm1-0/include/linux/rmap.h
--- linux-2.6.22-rc2-mm1.orig/include/linux/rmap.h 2007-05-29 18:01:57.000000000 +0400
+++ linux-2.6.22-rc2-mm1-0/include/linux/rmap.h 2007-05-30 16:13:09.000000000 +0400
@@ -69,9 +69,13 @@ void __anon_vma_link(struct vm_area_stru
  /*
   * rmap interfaces called when adding or removing pte of page
   */
-void page_add_anon_rmap(struct page *, struct vm_area_struct *, unsigned long);
-void page_add_new_anon_rmap(struct page *, struct vm_area_struct *, unsigned long);
-void page_add_file_rmap(struct page *);
+struct page_container;
+
+void page_add_anon_rmap(struct page *, struct vm_area_struct *,
+  unsigned long, struct page_container *);
+void page_add_new_anon_rmap(struct page *, struct vm_area_struct *,
+  unsigned long, struct page_container *);
+void page_add_file_rmap(struct page *, struct page_container *);
  void page_remove_rmap(struct page *, struct vm_area_struct *);

  #ifdef CONFIG_DEBUG_VM
diff -upr linux-2.6.22-rc2-mm1.orig/mm/memory.c linux-2.6.22-rc2-mm1-0/mm/memory.c
--- linux-2.6.22-rc2-mm1.orig/mm/memory.c 2007-05-30 12:32:36.000000000 +0400
+++ linux-2.6.22-rc2-mm1-0/mm/memory.c 2007-05-30 16:13:09.000000000 +0400
@@ -60,6 +60,8 @@
  #include <linux/swapops.h>
  #include <linux/elf.h>

+#include <linux/rss_container.h>
```

```
+
 #ifndef CONFIG_NEED_MULTIPLE_NODES
 /* use the per-pgdat data instead for discontigmem - mbligh */
 unsigned long max_mapnr;
@@ -1155,7 +1157,7 @@ static int zeromap_pte_range(struct mm_s
    break;
   }
   page_cache_get(page);
-  page_add_file_rmap(page);
+  page_add_file_rmap(page, NULL);
   inc_mm_counter(mm, file_rss);
   set_pte_at(mm, addr, pte, zero_pte);
  } while (pte++, addr += PAGE_SIZE, addr != end);
@@ -1263,7 +1265,7 @@ static int insert_page(struct mm_struct
  /* Ok, finally just insert the thing.. */
  get_page(page);
  inc_mm_counter(mm, file_rss);
- page_add_file_rmap(page);
+ page_add_file_rmap(page, NULL);
  set_pte_at(mm, addr, pte, mk_pte(page, prot));

  retval = 0;
@@ -1668,6 +1670,7 @@ static int do_wp_page(struct mm_struct *
  pte_t entry;
  int reuse = 0, ret = VM_FAULT_MINOR;
  struct page *dirty_page = NULL;
+ struct page_container *pcont;

  old_page = vm_normal_page(vma, address, orig_pte);
  if (!old_page)
@@ -1752,6 +1755,9 @@ gotten:
   cow_user_page(new_page, old_page, address, vma);
  }

+ if (container_rss_prepare(new_page, vma, &pcont))
+  goto oom;
+
  /*
   * Re-check the pte - we dropped the lock
   */
@@ -1779,12 +1785,14 @@ gotten:
   set_pte_at(mm, address, page_table, entry);
   update_mmu_cache(vma, address, entry);
   lru_cache_add_active(new_page);
-  page_add_new_anon_rmap(new_page, vma, address);
+  page_add_new_anon_rmap(new_page, vma, address, pcont);

   /* Free the old page.. */
```

```
   new_page = old_page;
   ret |= VM_FAULT_WRITE;
- }
+ } else
+  container_rss_release(pcont);
+
  if (new_page)
   page_cache_release(new_page);
  if (old_page)
@@ -2176,6 +2184,7 @@ static int do_swap_page(struct mm_struct
  swp_entry_t entry;
  pte_t pte;
  int ret = VM_FAULT_MINOR;
+ struct page_container *pcont;

  if (!pte_unmap_same(mm, pmd, page_table, orig_pte))
   goto out;
@@ -2208,6 +2217,11 @@ static int do_swap_page(struct mm_struct
   count_vm_event(PGMAJFAULT);
  }

+ if (container_rss_prepare(page, vma, &pcont)) {
+  ret = VM_FAULT_OOM;
+  goto out;
+ }
+
  delayacct_clear_flag(DELAYACCT_PF_SWAPIN);
  mark_page_accessed(page);
  lock_page(page);
@@ -2221,6 +2235,7 @@ static int do_swap_page(struct mm_struct

  if (unlikely(!PageUptodate(page))) {
   ret = VM_FAULT_SIGBUS;
+  container_rss_release(pcont);
   goto out_nomap;
  }

@@ -2235,7 +2250,7 @@ static int do_swap_page(struct mm_struct

  flush_icache_page(vma, page);
  set_pte_at(mm, address, page_table, pte);
- page_add_anon_rmap(page, vma, address);
+ page_add_anon_rmap(page, vma, address, pcont);

  swap_free(entry);
  if (vm_swap_full())
@@ -2256,6 +2271,7 @@ unlock:
 out:
```

```
  return ret;
 out_nomap:
+ container_rss_release(pcont);
  pte_unmap_unlock(page_table, ptl);
  unlock_page(page);
  page_cache_release(page);
@@ -2274,6 +2290,7 @@ static int do_anonymous_page(struct mm_s
  struct page *page;
  spinlock_t *ptl;
  pte_t entry;
+ struct page_container *pcont;

  if (write_access) {
   /* Allocate our own private page. */
@@ -2285,15 +2302,19 @@ static int do_anonymous_page(struct mm_s
   if (!page)
    goto oom;

+  if (container_rss_prepare(page, vma, &pcont))
+   goto oom_release;
+
   entry = mk_pte(page, vma->vm_page_prot);
   entry = maybe_mkwrite(pte_mkdirty(entry), vma);

   page_table = pte_offset_map_lock(mm, pmd, address, &ptl);
   if (!pte_none(*page_table))
-   goto release;
+   goto release_container;
+
   inc_mm_counter(mm, anon_rss);
   lru_cache_add_active(page);
-  page_add_new_anon_rmap(page, vma, address);
+  page_add_new_anon_rmap(page, vma, address, pcont);
  } else {
   /* Map the ZERO_PAGE - vm_page_prot is readonly */
   page = ZERO_PAGE(address);
@@ -2305,7 +2326,7 @@ static int do_anonymous_page(struct mm_s
   if (!pte_none(*page_table))
    goto release;
   inc_mm_counter(mm, file_rss);
-  page_add_file_rmap(page);
+  page_add_file_rmap(page, NULL);
  }

  set_pte_at(mm, address, page_table, entry);
@@ -2316,9 +2337,14 @@ static int do_anonymous_page(struct mm_s
 unlock:
  pte_unmap_unlock(page_table, ptl);
```

```
  return VM_FAULT_MINOR;
+release_container:
+ container_rss_release(pcont);
 release:
  page_cache_release(page);
  goto unlock;
+
+oom_release:
+ page_cache_release(page);
 oom:
  return VM_FAULT_OOM;
 }
@@ -2346,6 +2372,7 @@ static int __do_fault(struct mm_struct *
  int anon = 0;
  struct page *dirty_page = NULL;
  struct fault_data fdata;
+ struct page_container *pcont;

  fdata.address = address & PAGE_MASK;
  fdata.pgoff = pgoff;
@@ -2415,6 +2442,11 @@ static int __do_fault(struct mm_struct *

  }

+ if (container_rss_prepare(page, vma, &pcont)) {
+  fdata.type = VM_FAULT_OOM;
+  goto out;
+ }
+
  page_table = pte_offset_map_lock(mm, pmd, address, &ptl);

  /*
@@ -2437,10 +2467,10 @@ static int __do_fault(struct mm_struct *
   if (anon) {
                        inc_mm_counter(mm, anon_rss);
                        lru_cache_add_active(page);
-                       page_add_new_anon_rmap(page, vma, address);
+   page_add_new_anon_rmap(page, vma, address, pcont);
   } else {
    inc_mm_counter(mm, file_rss);
-   page_add_file_rmap(page);
+   page_add_file_rmap(page, pcont);
    if (flags & FAULT_FLAG_WRITE) {
     dirty_page = page;
     get_page(dirty_page);
@@ -2451,6 +2481,7 @@ static int __do_fault(struct mm_struct *
   update_mmu_cache(vma, address, entry);
   lazy_mmu_prot_update(entry);
```

```
  } else {
+  container_rss_release(pcont);
   if (anon)
    page_cache_release(page);
   else
diff -upr linux-2.6.22-rc2-mm1.orig/mm/migrate.c linux-2.6.22-rc2-mm1-0/mm/migrate.c
--- linux-2.6.22-rc2-mm1.orig/mm/migrate.c 2007-05-30 12:32:36.000000000 +0400
+++ linux-2.6.22-rc2-mm1-0/mm/migrate.c 2007-05-30 16:13:09.000000000 +0400
@@ -28,6 +28,7 @@
 #include <linux/mempolicy.h>
 #include <linux/vmalloc.h>
 #include <linux/security.h>
+#include <linux/rss_container.h>

 #include "internal.h"

@@ -134,6 +135,7 @@ static void remove_migration_pte(struct
  pte_t *ptep, pte;
   spinlock_t *ptl;
  unsigned long addr = page_address_in_vma(new, vma);
+ struct page_container *pcont;

  if (addr == -EFAULT)
   return;
@@ -157,6 +159,11 @@ static void remove_migration_pte(struct
    return;
   }

+ if (container_rss_prepare(new, vma, &pcont)) {
+  pte_unmap(ptep);
+  return;
+ }
+
   ptl = pte_lockptr(mm, pmd);
   spin_lock(ptl);
  pte = *ptep;
@@ -175,16 +182,19 @@ static void remove_migration_pte(struct
  set_pte_at(mm, addr, ptep, pte);

  if (PageAnon(new))
-  page_add_anon_rmap(new, vma, addr);
+  page_add_anon_rmap(new, vma, addr, pcont);
  else
-  page_add_file_rmap(new);
+  page_add_file_rmap(new, pcont);

  /* No need to invalidate - it was non-present before */
  update_mmu_cache(vma, addr, pte);
```

```
  lazy_mmu_prot_update(pte);
+ pte_unmap_unlock(ptep, ptl);
+ return;

 out:
  pte_unmap_unlock(ptep, ptl);
+ container_rss_release(pcont);
 }

 /*
diff -upr linux-2.6.22-rc2-mm1.orig/mm/page_alloc.c linux-2.6.22-rc2-mm1-0/mm/page_alloc.c
--- linux-2.6.22-rc2-mm1.orig/mm/page_alloc.c 2007-05-30 12:32:36.000000000 +0400
+++ linux-2.6.22-rc2-mm1-0/mm/page_alloc.c 2007-05-30 16:13:09.000000000 +0400
@@ -41,6 +41,7 @@
 #include <linux/pfn.h>
 #include <linux/backing-dev.h>
 #include <linux/fault-inject.h>
+#include <linux/rss_container.h>

 #include <asm/tlbflush.h>
 #include <asm/div64.h>
@@ -1042,6 +1043,7 @@ static void fastcall free_hot_cold_page(

  if (!PageHighMem(page))
   debug_check_no_locks_freed(page_address(page), PAGE_SIZE);
+ init_page_container(page);
  arch_free_page(page, 0);
  kernel_map_pages(page, 1, 0);

@@ -2592,6 +2594,7 @@ void __meminit memmap_init_zone(unsigned
   set_page_links(page, zone, nid, pfn);
   init_page_count(page);
   reset_page_mapcount(page);
+  init_page_container(page);
   SetPageReserved(page);

   /*
diff -upr linux-2.6.22-rc2-mm1.orig/mm/rmap.c linux-2.6.22-rc2-mm1-0/mm/rmap.c
--- linux-2.6.22-rc2-mm1.orig/mm/rmap.c 2007-05-30 12:32:36.000000000 +0400
+++ linux-2.6.22-rc2-mm1-0/mm/rmap.c 2007-05-30 16:13:09.000000000 +0400
@@ -51,6 +51,8 @@

 #include <asm/tlbflush.h>

+#include <linux/rss_container.h>
+
 struct kmem_cache *anon_vma_cachep;
```

```
 static inline void validate_anon_vma(struct vm_area_struct *find_vma)
@@ -563,18 +565,23 @@ static void __page_check_anon_rmap(struc
  * @page: the page to add the mapping to
  * @vma: the vm area in which the mapping is added
  * @address: the user virtual address mapped
+ * @pcont: the page beancounter to charge page with
  *
  * The caller needs to hold the pte lock and the page must be locked.
  */
 void page_add_anon_rmap(struct page *page,
- struct vm_area_struct *vma, unsigned long address)
+ struct vm_area_struct *vma, unsigned long address,
+ struct page_container *pcont)
 {
  VM_BUG_ON(!PageLocked(page));
  VM_BUG_ON(address < vma->vm_start || address >= vma->vm_end);
- if (atomic_inc_and_test(&page->_mapcount))
+ if (atomic_inc_and_test(&page->_mapcount)) {
+  container_rss_add(pcont);
   __page_set_anon_rmap(page, vma, address);
- else
+ } else {
   __page_check_anon_rmap(page, vma, address);
+  container_rss_release(pcont);
+ }
 }

 /*
@@ -582,29 +589,37 @@ void page_add_anon_rmap(struct page *pag
  * @page: the page to add the mapping to
  * @vma: the vm area in which the mapping is added
  * @address: the user virtual address mapped
+ * @pcont: the page beancounter to charge page with
  *
  * Same as page_add_anon_rmap but must only be called on *new* pages.
  * This means the inc-and-test can be bypassed.
  * Page does not have to be locked.
  */
 void page_add_new_anon_rmap(struct page *page,
- struct vm_area_struct *vma, unsigned long address)
+ struct vm_area_struct *vma, unsigned long address,
+ struct page_container *pcont)
 {
  BUG_ON(address < vma->vm_start || address >= vma->vm_end);
  atomic_set(&page->_mapcount, 0); /* elevate count by 1 (starts at -1) */
+ container_rss_add(pcont);
  __page_set_anon_rmap(page, vma, address);
 }
```

```
 /**
  * page_add_file_rmap - add pte mapping to a file page
- * @page: the page to add the mapping to
+ * @page: the page to add the mapping to
+ * @pcont: the page beancounter to charge page with
  *
  * The caller needs to hold the pte lock.
  */
-void page_add_file_rmap(struct page *page)
+void page_add_file_rmap(struct page *page, struct page_container *pcont)
 {
- if (atomic_inc_and_test(&page->_mapcount))
+ if (atomic_inc_and_test(&page->_mapcount)) {
+  if (pcont)
+   container_rss_add(pcont);
   __inc_zone_page_state(page, NR_FILE_MAPPED);
+ } else if (pcont)
+  container_rss_release(pcont);
 }

 #ifdef CONFIG_DEBUG_VM
@@ -635,6 +650,9 @@ void page_dup_rmap(struct page *page, st
  */
 void page_remove_rmap(struct page *page, struct vm_area_struct *vma)
 {
+ struct page_container *pcont;
+
+ pcont = page_container(page);
  if (atomic_add_negative(-1, &page->_mapcount)) {
   if (unlikely(page_mapcount(page) < 0)) {
    printk (KERN_EMERG "Eeek! page_mapcount(page) went negative! (%d)\n",
page_mapcount(page));
@@ -652,6 +670,8 @@ void page_remove_rmap(struct page *page,
    BUG();
   }

+  if (pcont)
+   container_rss_del(pcont);
   /*
    * It would be tidy to reset the PageAnon mapping here,
    * but that might overwrite a racing page_add_anon_rmap
diff -upr linux-2.6.22-rc2-mm1.orig/mm/swapfile.c linux-2.6.22-rc2-mm1-0/mm/swapfile.c
--- linux-2.6.22-rc2-mm1.orig/mm/swapfile.c 2007-05-11 16:36:58.000000000 +0400
+++ linux-2.6.22-rc2-mm1-0/mm/swapfile.c 2007-05-30 16:13:09.000000000 +0400
@@ -32,6 +32,8 @@
 #include <asm/tlbflush.h>
 #include <linux/swapops.h>
```

```
+#include <linux/rss_container.h>
+
 DEFINE_SPINLOCK(swap_lock);
 unsigned int nr_swapfiles;
 long total_swap_pages;
@@ -507,13 +509,14 @@ unsigned int count_swap_pages(int type,
  * force COW, vm_page_prot omits write permission from any private vma.
  */
 static void unuse_pte(struct vm_area_struct *vma, pte_t *pte,
-  unsigned long addr, swp_entry_t entry, struct page *page)
+  unsigned long addr, swp_entry_t entry, struct page *page,
+  struct page_container *pcont)
 {
  inc_mm_counter(vma->vm_mm, anon_rss);
  get_page(page);
  set_pte_at(vma->vm_mm, addr, pte,
     pte_mkold(mk_pte(page, vma->vm_page_prot)));
- page_add_anon_rmap(page, vma, addr);
+ page_add_anon_rmap(page, vma, addr, pcont);
  swap_free(entry);
  /*
   * Move the page to the active list so it is not
@@ -530,6 +533,10 @@ static int unuse_pte_range(struct vm_are
  pte_t *pte;
  spinlock_t *ptl;
  int found = 0;
+ struct page_container *pcont;
+
+ if (container_rss_prepare(page, vma, &pcont))
+  return 0;

  pte = pte_offset_map_lock(vma->vm_mm, pmd, addr, &ptl);
  do {
@@ -538,12 +545,14 @@ static int unuse_pte_range(struct vm_are
    * Test inline before going to call unuse_pte.
    */
   if (unlikely(pte_same(*pte, swp_pte))) {
-   unuse_pte(vma, pte++, addr, entry, page);
+   unuse_pte(vma, pte++, addr, entry, page, pcont);
    found = 1;
    break;
   }
  } while (pte++, addr += PAGE_SIZE, addr != end);
  pte_unmap_unlock(pte - 1, ptl);
+ if (!found)
+  container_rss_release(pcont);
  return found;
```

}

Subject: [PATCH 6/8] Per container OOM killer
Posted by Pavel Emelianov on Wed, 30 May 2007 15:31:12 GMT
View Forum Message <> Reply to Message

When container is completely out of memory some tasks should die.
This is unfair to kill the current task, so a task with the largest
RSS is chosen and killed. The code re-uses current OOM killer
select_bad_process() for task selection.

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

---

```
diff -upr linux-2.6.22-rc2-mm1.orig/include/linux/rss_container.h
linux-2.6.22-rc2-mm1-0/include/linux/rss_container.h
--- linux-2.6.22-rc2-mm1.orig/include/linux/rss_container.h 2007-05-30 16:16:58.000000000
+0400
+++ linux-2.6.22-rc2-mm1-0/include/linux/rss_container.h 2007-05-30 16:13:09.000000000 +0400
@@ -10,6 +10,7 @@
 void container_rss_add(struct page_container *);
 void container_rss_del(struct page_container *);
 void container_rss_release(struct page_container *);
+void container_out_of_memory(struct rss_container *);

 void mm_init_container(struct mm_struct *mm, struct task_struct *tsk);
 void mm_free_container(struct mm_struct *mm);
diff -upr linux-2.6.22-rc2-mm1.orig/mm/oom_kill.c linux-2.6.22-rc2-mm1-0/mm/oom_kill.c
--- linux-2.6.22-rc2-mm1.orig/mm/oom_kill.c 2007-05-11 16:36:58.000000000 +0400
+++ linux-2.6.22-rc2-mm1-0/mm/oom_kill.c 2007-05-30 16:13:09.000000000 +0400
@@ -24,6 +24,7 @@
 #include <linux/cpuset.h>
 #include <linux/module.h>
 #include <linux/notifier.h>
+#include <linux/rss_container.h>

 int sysctl_panic_on_oom;
 /* #define DEBUG */
@@ -47,7 +48,8 @@ int sysctl_panic_on_oom;
  *    of least surprise ... (be careful when you change it)
  */

-unsigned long badness(struct task_struct *p, unsigned long uptime)
+unsigned long badness(struct task_struct *p, unsigned long uptime,
+  struct rss_container *rss)
 {
```

```
  unsigned long points, cpu_time, run_time, s;
  struct mm_struct *mm;
@@ -60,6 +62,13 @@ unsigned long badness(struct task_struct
   return 0;
  }

+#ifdef CONFIG_RSS_CONTAINER
+ if (rss != NULL && mm->rss_container != rss) {
+  task_unlock(p);
+  return 0;
+ }
+#endif
+
  /*
   * The memory size of the process is the basis for the badness.
   */
@@ -204,7 +213,8 @@ static inline int constrained_alloc(stru
  *
  * (not docbooked, we don't want this one cluttering up the manual)
  */
-static struct task_struct *select_bad_process(unsigned long *ppoints)
+static struct task_struct *select_bad_process(unsigned long *ppoints,
+  struct rss_container *rss)
 {
  struct task_struct *g, *p;
  struct task_struct *chosen = NULL;
@@ -258,7 +268,7 @@ static struct task_struct *select_bad_pr
   if (p->oomkilladj == OOM_DISABLE)
    continue;

-  points = badness(p, uptime.tv_sec);
+  points = badness(p, uptime.tv_sec, rss);
   if (points > *ppoints || !chosen) {
    chosen = p;
    *ppoints = points;
@@ -444,7 +454,7 @@ retry:
    * Rambo mode: Shoot down a process and hope it solves whatever
    * issues we may have.
    */
-  p = select_bad_process(&points);
+  p = select_bad_process(&points, NULL);

   if (PTR_ERR(p) == -1UL)
    goto out;
@@ -473,3 +483,27 @@ out:
  if (!test_thread_flag(TIF_MEMDIE))
   schedule_timeout_uninterruptible(1);
 }
```

```
+
+#ifdef CONFIG_RSS_CONTAINER
+void container_out_of_memory(struct rss_container *rss)
+{
+ unsigned long points = 0;
+ struct task_struct *p;
+
+ container_lock();
+ read_lock(&tasklist_lock);
+retry:
+ p = select_bad_process(&points, rss);
+ if (PTR_ERR(p) == -1UL)
+  goto out;
+
+ if (!p)
+  p = current;
+
+ if (oom_kill_process(p, points, "Container out of memory"))
+  goto retry;
+out:
+ read_unlock(&tasklist_lock);
+ container_unlock();
+}
+#endif
 void mm_free_container(struct mm_struct *mm);
```

---

Subject: [PATCH 7/8] Scanner changes needed to implement per-container scanner
Posted by Pavel Emelianov on Wed, 30 May 2007 15:35:20 GMT
View Forum Message <> Reply to Message

The core change is that the isolate_lru_pages() call is
replaced with struct scan_controll->isolate_pages() call.

Other changes include exporting __isolate_lru_page() for
per-container isolator and handling variable-to-pointer
changes in try_to_free_pages().

This makes possible to use different isolation routines
for per-container page reclamation. This will be used by
the following patch.

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

---

diff -upr linux-2.6.22-rc2-mm1.orig/include/linux/swap.h
linux-2.6.22-rc2-mm1-0/include/linux/swap.h

```
--- linux-2.6.22-rc2-mm1.orig/include/linux/swap.h 2007-05-30 12:32:36.000000000 +0400
+++ linux-2.6.22-rc2-mm1-0/include/linux/swap.h 2007-05-30 16:13:09.000000000 +0400
@@ -192,6 +192,11 @@ extern void swap_setup(void);
 /* linux/mm/vmscan.c */
 extern unsigned long try_to_free_pages(struct zone **zones, int order,
      gfp_t gfp_mask);
+
+struct rss_container;
+extern unsigned long try_to_free_pages_in_container(struct rss_container *);
+int __isolate_lru_page(struct page *page, int mode);
+
 extern unsigned long shrink_all_memory(unsigned long nr_pages);
 extern int vm_swappiness;
 extern int remove_mapping(struct address_space *mapping, struct page *page);
diff -upr linux-2.6.22-rc2-mm1.orig/mm/vmscan.c linux-2.6.22-rc2-mm1-0/mm/vmscan.c
--- linux-2.6.22-rc2-mm1.orig/mm/vmscan.c 2007-05-30 12:32:36.000000000 +0400
+++ linux-2.6.22-rc2-mm1-0/mm/vmscan.c 2007-05-30 16:13:09.000000000 +0400
@@ -47,6 +47,8 @@

 #include "internal.h"

+#include <linux/rss_container.h>
+
 struct scan_control {
  /* Incremented by the number of inactive pages that were scanned */
  unsigned long nr_scanned;
@@ -70,6 +72,13 @@ struct scan_control {
  int all_unreclaimable;

  int order;
+
+ struct rss_container *cnt;
+
+ unsigned long (*isolate_pages)(unsigned long nr, struct list_head *dst,
+    unsigned long *scanned, int order, int mode,
+    struct zone *zone, struct rss_container *cont,
+    int active);
 };

 #define lru_to_page(_head) (list_entry((_head)->prev, struct page, lru))
@@ -622,7 +631,7 @@ keep:
  *
  * returns 0 on success, -ve errno on failure.
  */
-static int __isolate_lru_page(struct page *page, int mode)
+int __isolate_lru_page(struct page *page, int mode)
 {
  int ret = -EINVAL;
```

```
@@ -774,6 +783,19 @@ static unsigned long clear_active_flags(
  return nr_active;
 }

+static unsigned long isolate_pages_global(unsigned long nr,
+  struct list_head *dst, unsigned long *scanned,
+  int order, int mode, struct zone *z,
+  struct rss_container *cont, int active)
+{
+ if (active)
+  return isolate_lru_pages(nr, &z->active_list,
+    dst, scanned, order, mode);
+ else
+  return isolate_lru_pages(nr, &z->inactive_list,
+    dst, scanned, order, mode);
+}
+
 /*
  * shrink_inactive_list() is a helper for shrink_zone().  It returns the number
  * of reclaimed pages
@@ -797,11 +819,11 @@ static unsigned long shrink_inactive_lis
   unsigned long nr_freed;
   unsigned long nr_active;

-  nr_taken = isolate_lru_pages(sc->swap_cluster_max,
-        &zone->inactive_list,
-        &page_list, &nr_scan, sc->order,
-        (sc->order > PAGE_ALLOC_COSTLY_ORDER)?
-         ISOLATE_BOTH : ISOLATE_INACTIVE);
+  nr_taken = sc->isolate_pages(sc->swap_cluster_max, &page_list,
+    &nr_scan, sc->order,
+    (sc->order > PAGE_ALLOC_COSTLY_ORDER) ?
+     ISOLATE_BOTH : ISOLATE_INACTIVE,
+    zone, sc->cnt, 0);
   nr_active = clear_active_flags(&page_list);

   __mod_zone_page_state(zone, NR_ACTIVE, -nr_active);
@@ -950,8 +972,8 @@ force_reclaim_mapped:

  lru_add_drain();
  spin_lock_irq(&zone->lru_lock);
- pgmoved = isolate_lru_pages(nr_pages, &zone->active_list,
-       &l_hold, &pgscanned, sc->order, ISOLATE_ACTIVE);
+ pgmoved = sc->isolate_pages(nr_pages, &l_hold, &pgscanned,
+   sc->order, ISOLATE_ACTIVE, zone, sc->cnt, 1);
  zone->pages_scanned += pgscanned;
  __mod_zone_page_state(zone, NR_ACTIVE, -pgmoved);
```

```
  spin_unlock_irq(&zone->lru_lock);
@@ -1142,7 +1166,8 @@ static unsigned long shrink_zones(int pr
  * holds filesystem locks which prevent writeout this might not work, and the
  * allocation attempt will fail.
  */
-unsigned long try_to_free_pages(struct zone **zones, int order, gfp_t gfp_mask)
+unsigned long do_try_to_free_pages(struct zone **zones, gfp_t gfp_mask,
+  struct scan_control *sc)
 {
  int priority;
  int ret = 0;
@@ -1151,14 +1176,6 @@ unsigned long try_to_free_pages(struct z
  struct reclaim_state *reclaim_state = current->reclaim_state;
  unsigned long lru_pages = 0;
  int i;
- struct scan_control sc = {
-  .gfp_mask = gfp_mask,
-  .may_writepage = !laptop_mode,
-  .swap_cluster_max = SWAP_CLUSTER_MAX,
-  .may_swap = 1,
-  .swappiness = vm_swappiness,
-  .order = order,
- };

  delay_swap_prefetch();
  count_vm_event(ALLOCSTALL);
@@ -1174,17 +1191,18 @@ unsigned long try_to_free_pages(struct z
  }

  for (priority = DEF_PRIORITY; priority >= 0; priority--) {
-  sc.nr_scanned = 0;
+  sc->nr_scanned = 0;
   if (!priority)
    disable_swap_token();
-  nr_reclaimed += shrink_zones(priority, zones, &sc);
-  shrink_slab(sc.nr_scanned, gfp_mask, lru_pages);
+  nr_reclaimed += shrink_zones(priority, zones, sc);
+  if (sc->cnt == NULL)
+   shrink_slab(sc->nr_scanned, gfp_mask, lru_pages);
   if (reclaim_state) {
    nr_reclaimed += reclaim_state->reclaimed_slab;
    reclaim_state->reclaimed_slab = 0;
   }
-  total_scanned += sc.nr_scanned;
-  if (nr_reclaimed >= sc.swap_cluster_max) {
+  total_scanned += sc->nr_scanned;
+  if (nr_reclaimed >= sc->swap_cluster_max) {
    ret = 1;
```

```
    goto out;
   }
@@ -1196,18 +1214,18 @@ unsigned long try_to_free_pages(struct z
    * that's undesirable in laptop mode, where we *want* lumpy
    * writeout.  So in laptop mode, write out the whole world.
    */
-  if (total_scanned > sc.swap_cluster_max +
-     sc.swap_cluster_max / 2) {
+  if (total_scanned > sc->swap_cluster_max +
+      sc->swap_cluster_max / 2) {
    wakeup_pdflush(laptop_mode ? 0 : total_scanned);
-   sc.may_writepage = 1;
+   sc->may_writepage = 1;
   }

   /* Take a nap, wait for some writeback to complete */
-  if (sc.nr_scanned && priority < DEF_PRIORITY - 2)
+  if (sc->nr_scanned && priority < DEF_PRIORITY - 2)
    congestion_wait(WRITE, HZ/10);
  }
  /* top priority shrink_caches still had more to do? don't OOM, then */
- if (!sc.all_unreclaimable)
+ if (!sc->all_unreclaimable && sc->cnt == NULL)
   ret = 1;
 out:
  /*
@@ -1230,6 +1248,22 @@ out:
  return ret;
 }

+unsigned long try_to_free_pages(struct zone **zones, int order, gfp_t gfp_mask)
+{
+ struct scan_control sc = {
+  .gfp_mask = gfp_mask,
+  .may_writepage = !laptop_mode,
+  .swap_cluster_max = SWAP_CLUSTER_MAX,
+  .may_swap = 1,
+  .swappiness = vm_swappiness,
+  .order = order,
+  .cnt = NULL,
+  .isolate_pages = isolate_pages_global,
+ };
+
+ return do_try_to_free_pages(zones, gfp_mask, &sc);
+}
+
 /*
  * For kswapd, balance_pgdat() will work across all this node's zones until
```

```
  * they are all at pages_high.
@@ -1265,6 +1329,8 @@ static unsigned long balance_pgdat(pg_da
   .swap_cluster_max = SWAP_CLUSTER_MAX,
   .swappiness = vm_swappiness,
   .order = order,
+  .cnt = NULL,
+  .isolate_pages = isolate_pages_global,
  };
  /*
   * temp_priority is used to remember the scanning priority at which
@@ -1604,6 +1670,8 @@ unsigned long shrink_all_memory(unsigned
   .swap_cluster_max = nr_pages,
   .may_writepage = 1,
   .swappiness = vm_swappiness,
+  .cnt = NULL,
+  .isolate_pages = isolate_pages_global,
  };

  delay_swap_prefetch();
@@ -1789,6 +1857,8 @@ static int __zone_reclaim(struct zone *z
      SWAP_CLUSTER_MAX),
   .gfp_mask = gfp_mask,
   .swappiness = vm_swappiness,
+  .cnt = NULL,
+  .isolate_pages = isolate_pages_global,
  };
  unsigned long slab_reclaimable;
```

---

Subject: [PATCH 8/8] Per-container pages reclamation
Posted by Pavel Emelianov on Wed, 30 May 2007 15:38:01 GMT
View Forum Message <> Reply to Message

Implement try_to_free_pages_in_container() to free the
pages in container that has run out of memory.

The scan_control->isolate_pages() function is set to
isolate_pages_in_container() that isolates the container
pages only. The exported __isolate_lru_page() call
makes things look simpler than in the previous version.

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

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

---

diff -upr linux-2.6.22-rc2-mm1.orig/mm/rss_container.c

```
linux-2.6.22-rc2-mm1-0/mm/rss_container.c
--- linux-2.6.22-rc2-mm1.orig/mm/rss_container.c 2007-05-30 16:16:58.000000000 +0400
+++ linux-2.6.22-rc2-mm1-0/mm/rss_container.c 2007-05-30 16:13:09.000000000 +0400
@@ -135,6 +135,76 @@
  spin_unlock(&rss->res.lock);
 }

+void container_rss_move_lists(struct page *pg, bool active)
+{
+ struct rss_container *rss;
+ struct page_container *pc;
+
+ if (!page_mapped(pg))
+  return;
+
+ pc = page_container(pg);
+ if (pc == NULL)
+  return;
+
+ rss = pc->cnt;
+
+ spin_lock(&rss->res.lock);
+ if (active)
+  list_move(&pc->list, &rss->active_list);
+ else
+  list_move(&pc->list, &rss->inactive_list);
+ spin_unlock(&rss->res.lock);
+}
+
+static unsigned long isolate_container_pages(unsigned long nr_to_scan,
+  struct list_head *src, struct list_head *dst,
+  unsigned long *scanned, struct zone *zone, int mode)
+{
+ unsigned long nr_taken = 0;
+ struct page *page;
+ struct page_container *pc;
+ unsigned long scan;
+ LIST_HEAD(pc_list);
+
+ for (scan = 0; scan < nr_to_scan && !list_empty(src); scan++) {
+  pc = list_entry(src->prev, struct page_container, list);
+  page = pc->page;
+  if (page_zone(page) != zone)
+   continue;
+
+  list_move(&pc->list, &pc_list);
+
+  if (__isolate_lru_page(page, mode) == 0) {
```

```
+   list_move(&page->lru, dst);
+   nr_taken++;
+  }
+ }
+
+ list_splice(&pc_list, src);
+
+ *scanned = scan;
+ return nr_taken;
+}
+
+unsigned long isolate_pages_in_container(unsigned long nr_to_scan,
+  struct list_head *dst, unsigned long *scanned,
+  int order, int mode, struct zone *zone,
+  struct rss_container *rss, int active)
+{
+ unsigned long ret;
+
+ spin_lock(&rss->res.lock);
+ if (active)
+  ret = isolate_container_pages(nr_to_scan, &rss->active_list,
+    dst, scanned, zone, mode);
+ else
+  ret = isolate_container_pages(nr_to_scan, &rss->inactive_list,
+    dst, scanned, zone, mode);
+ spin_unlock(&rss->res.lock);
+ return ret;
+}
+
 void container_rss_add(struct page_container *pc)
 {
  struct page *pg;
diff -upr linux-2.6.22-rc2-mm1.orig/mm/swap.c linux-2.6.22-rc2-mm1-0/mm/swap.c
--- linux-2.6.22-rc2-mm1.orig/mm/swap.c 2007-05-30 12:32:36.000000000 +0400
+++ linux-2.6.22-rc2-mm1-0/mm/swap.c 2007-05-30 16:13:09.000000000 +0400
@@ -31,6 +31,7 @@
 #include <linux/cpu.h>
 #include <linux/notifier.h>
 #include <linux/init.h>
+#include <linux/rss_container.h>

 /* How many pages do we try to swap or page in/out together? */
 int page_cluster;
@@ -148,6 +149,7 @@ void fastcall activate_page(struct page
   SetPageActive(page);
   add_page_to_active_list(zone, page);
   __count_vm_event(PGACTIVATE);
+  container_rss_move_lists(page, 1);
```

```
  }
  spin_unlock_irq(&zone->lru_lock);
 }
diff -upr linux-2.6.22-rc2-mm1.orig/include/linux/rss_container.h
linux-2.6.22-rc2-mm1-0/include/linux/rss_container.h
--- linux-2.6.22-rc2-mm1.orig/include/linux/rss_container.h 2007-05-30 16:16:58.000000000
+0400
+++ linux-2.6.22-rc2-mm1-0/include/linux/rss_container.h 2007-05-30 16:13:09.000000000 +0400
@@ -24,6 +24,11 @@
 void mm_init_container(struct mm_struct *mm, struct task_struct *tsk);
 void mm_free_container(struct mm_struct *mm);

+void container_rss_move_lists(struct page *pg, bool active);
+unsigned long isolate_pages_in_container(unsigned long nr_to_scan,
+  struct list_head *dst, unsigned long *scanned,
+  int order, int mode, struct zone *zone,
+  struct rss_container *, int active);
 static inline void init_page_container(struct page *page)
 {
  page_container(page) = NULL;
@@ -64,5 +68,7 @@
 {
 }

+#define isolate_container_pages(nr, dst, scanned, rss, act, zone) ({ BUG(); 0;})
+#define container_rss_move_lists(pg, active) do { } while (0)
 #endif
 #endif
diff -upr linux-2.6.22-rc2-mm1.orig/mm/vmscan.c linux-2.6.22-rc2-mm1-0/mm/vmscan.c
--- linux-2.6.22-rc2-mm1.orig/mm/vmscan.c 2007-05-30 12:32:36.000000000 +0400
+++ linux-2.6.22-rc2-mm1-0/mm/vmscan.c 2007-05-30 16:13:09.000000000 +0400
@@ -983,6 +1005,7 @@ force_reclaim_mapped:
   ClearPageActive(page);

   list_move(&page->lru, &zone->inactive_list);
+  container_rss_move_lists(page, 0);
   pgmoved++;
   if (!pagevec_add(&pvec, page)) {
    __mod_zone_page_state(zone, NR_INACTIVE, pgmoved);
@@ -1011,6 +1034,7 @@ force_reclaim_mapped:
   SetPageLRU(page);
   VM_BUG_ON(!PageActive(page));
   list_move(&page->lru, &zone->active_list);
+  container_rss_move_lists(page, 1);
   pgmoved++;
   if (!pagevec_add(&pvec, page)) {
    __mod_zone_page_state(zone, NR_ACTIVE, pgmoved);
@@ -1230,6 +1248,36 @@ out:
```

```
  return ret;
 }

+#ifdef CONFIG_RSS_CONTAINER
+unsigned long try_to_free_pages_in_container(struct rss_container *cnt)
+{
+ struct scan_control sc = {
+  .gfp_mask = GFP_KERNEL,
+  .may_writepage = 1,
+  .swap_cluster_max = 1,
+  .may_swap = 1,
+  .swappiness = vm_swappiness,
+  .order = 0, /* in this case we wanted one page only */
+  .cnt = cnt,
+  .isolate_pages = isolate_pages_in_container,
+ };
+ int node;
+ struct zone **zones;
+
+ for_each_online_node(node) {
+#ifdef CONFIG_HIGHMEM
+  zones = NODE_DATA(node)->node_zonelists[ZONE_HIGHMEM].zones;
+  if (do_try_to_free_pages(zones, sc.gfp_mask, &sc))
+   return 1;
+#endif
+  zones = NODE_DATA(node)->node_zonelists[ZONE_NORMAL].zones;
+  if (do_try_to_free_pages(zones, sc.gfp_mask, &sc))
+   return 1;
+ }
+ return 0;
+}
+#endif
+
 unsigned long try_to_free_pages(struct zone **zones, int order, gfp_t gfp_mask)
 {
  struct scan_control sc = {
```

---

Subject: Re: [PATCH 4/8] RSS container core
Posted by Andrew Morton on Wed, 30 May 2007 21:46:17 GMT
View Forum Message <> Reply to Message

On Wed, 30 May 2007 19:32:14 +0400
Pavel Emelianov <xemul@openvz.org> wrote:

> The core routines for tracking the page ownership,
> registration of RSS subsystem in the containers and
> the definition of rss_container as container subsystem

> combined   with resource counter.
>
> --- linux-2.6.22-rc2-mm1.orig/include/linux/rss_container.h 2007-05-30 16:16:58.000000000
+0400
> +++ linux-2.6.22-rc2-mm1-0/include/linux/rss_container.h 2007-05-30 16:13:09.000000000
+0400
> @@ -0,0 +1,64 @@
> +#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 *);
> +
> +void mm_init_container(struct mm_struct *mm, struct task_struct *tsk);
> +void mm_free_container(struct mm_struct *mm);
> +
> +static inline void init_page_container(struct page *page)
> +{
> + page_container(page) = NULL;
> +}

Please, no.  Using a "function" as an lvalue like this _requires_ that the
"function" be implemented as a macro, which rather defeats the whole point.

Much nicer to do

```
#ifdef CONFIG_FOO
static inline void set_page_container(struct page *page,
     struct page_container *page_container)
{
 page->rss_container = page_container;
}
```

```
#else
static inline void set_page_container(struct page *page,
     struct page_container *page_container)
{
}
#endif

> +#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 */
```

eh?  What is gcc's problem here?

```
> + return 0;
> +}
>
> ...
>
> @@ -0,0 +1,271 @@
> +/*
> + * 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/swap.h>
> +#include <linux/res_counter.h>
> +#include <linux/rss_container.h>
> +
> +struct rss_container {
> + struct res_counter res;
> + struct list_head inactive_list;
> + struct list_head active_list;
> + atomic_t rss_reclaimed;
> + struct container_subsys_state css;
> +};
> +
> +struct page_container {
> + struct page *page;
> + struct rss_container *cnt;
```

> + struct list_head list;
> +};

Document each member carefully, please.  This is where your readers will
first come when trying to understand your code.

> +}
> +
> +int container_rss_prepare(struct page *page, struct vm_area_struct *vma,
> +  struct page_container **ppc)
> +{

This is an important-looking kernel-wide function.  It needs a nice comment
telling people what its role in life is.

> + 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;
> +
> + while (res_counter_charge(&rss->res, 1)) {
> +  if (try_to_free_pages_in_container(rss)) {
> +   atomic_inc(&rss->rss_reclaimed);
> +   continue;
> +  }

I find it mysterious that rss->rss_reclaimed gets incremented by one when
try_to_free_pages_in_container() succeeds (or is it when it fails?
try_to_free_pages_in_container() is undocumented, so it's hard to tell).
rss_reclaimed is also undocumented, adding to the mystery.

Also, try_to_free_pages_in_container() gets added in a later patch.  The
whole series seems backwards.  Probably this problem is less serious, as
things will still build and run when CONFIG_RSS_CONTAINER=n

> +  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_irq(&rss->res.lock);
> + list_add(&pc->list, &rss->active_list);
> + spin_unlock_irq(&rss->res.lock);
> +
> + 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);
> +}

Please document all the above.

> +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;

I cannot tell what would cause the above test-and-goto to be taken.
Please add a comment describing this piece of code.

> + 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 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);
> +}
> +
> +
> +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,
> +};
```

Did we add user documentation for the above?

```
> +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;

If we fail partway through here, do the thus-far-created fiels get cleaned up?

> + 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,
> +};

Did this need kernel-wide scope?

---

Subject: Re: [PATCH 5/8] RSS accounting hooks over the code
Posted by Andrew Morton on Wed, 30 May 2007 21:46:29 GMT
View Forum Message <> Reply to Message

On Wed, 30 May 2007 19:34:18 +0400
Pavel Emelianov <xemul@openvz.org> wrote:

> As described above, pages are charged to their first touchers.
> The first toucher is determined using pages' _mapcount
> manipulations in rmap calls.
>
> Page is charged in two stages:
> 1. preparation, in which the resource availability is checked.
>    This stage may lead to page reclamation, thus it is performed
>    in a "might-sleep" places;
> 2. the container assignment to page. This is done in an atomic
>    code that handles races between multiple touchers.

I suppose we need to think about what to do about higher-order pages, and
compound pages, and hugetlb memory.

---

Subject: Re: [PATCH 7/8] Scanner changes needed to implement per-container scanner
Posted by Andrew Morton on Wed, 30 May 2007 21:46:47 GMT
View Forum Message <> Reply to Message

On Wed, 30 May 2007 19:39:41 +0400
Pavel Emelianov <xemul@openvz.org> wrote:

> The core change is that the isolate_lru_pages() call is
> replaced with struct scan_controll->isolate_pages() call.
>
> Other changes include exporting __isolate_lru_page() for
> per-container isolator and handling variable-to-pointer
> changes in try_to_free_pages().
>
> This makes possible to use different isolation routines
> for per-container page reclamation. This will be used by
> the following patch.
>
> ...
>
> +struct rss_container;
> +extern unsigned long try_to_free_pages_in_container(struct rss_container *);
> +int __isolate_lru_page(struct page *page, int mode);


>  extern unsigned long shrink_all_memory(unsigned long nr_pages);
>  extern int vm_swappiness;
>  extern int remove_mapping(struct address_space *mapping, struct page *page);
> diff -upr linux-2.6.22-rc2-mm1.orig/mm/vmscan.c linux-2.6.22-rc2-mm1-0/mm/vmscan.c
> --- linux-2.6.22-rc2-mm1.orig/mm/vmscan.c 2007-05-30 12:32:36.000000000 +0400
> +++ linux-2.6.22-rc2-mm1-0/mm/vmscan.c 2007-05-30 16:13:09.000000000 +0400
> @@ -47,6 +47,8 @@
>
>  #include "internal.h"
>
> +#include <linux/rss_container.h>
> +
>  struct scan_control {
>   /* Incremented by the number of inactive pages that were scanned */
>   unsigned long nr_scanned;
> @@ -70,6 +72,13 @@ struct scan_control {
>   int all_unreclaimable;
>

>   int order;
> +
> + struct rss_container *cnt;

Can we please have a better name?  "cnt" is usually a (poorly-chosen) name
for an integer counter.  Perhaps "container", or even "rss_container".

> +  nr_reclaimed += shrink_zones(priority, zones, sc);
> +  if (sc->cnt == NULL)
> +   shrink_slab(sc->nr_scanned, gfp_mask, lru_pages);

We don't we shrink slab if called to shrink a container.

This is a fundamental design decision, and a design shortcoming.  A full
discussion of this is absolutely appropriate to the patch changelog.
Please don't just hide stuff like this in the patch and leave people
wondering, or ignorant.

---

Subject: Re: [PATCH 8/8] Per-container pages reclamation
Posted by Andrew Morton on Wed, 30 May 2007 21:47:37 GMT
View Forum Message <> Reply to Message

On Wed, 30 May 2007 19:42:26 +0400
Pavel Emelianov <xemul@openvz.org> wrote:

> Implement try_to_free_pages_in_container() to free the
> pages in container that has run out of memory.
>
> The scan_control->isolate_pages() function is set to
> isolate_pages_in_container() that isolates the container
> pages only. The exported __isolate_lru_page() call
> makes things look simpler than in the previous version.
>
> Includes fix from Balbir Singh <balbir@in.ibm.com>
>
>  }
>
> +void container_rss_move_lists(struct page *pg, bool active)
> +{
> + struct rss_container *rss;
> + struct page_container *pc;
> +
> + if (!page_mapped(pg))
> +  return;
> +
> + pc = page_container(pg);
> + if (pc == NULL)

> +  return;
> +
> + rss = pc->cnt;
> +
> + spin_lock(&rss->res.lock);
> + if (active)
> +  list_move(&pc->list, &rss->active_list);
> + else
> +  list_move(&pc->list, &rss->inactive_list);
> + spin_unlock(&rss->res.lock);
> +}

This is an interesting-looking function.  Please document it?

I'm inferring that the rss container has an active and inactive list and
that this basically follows the same operation as the traditional per-zone
lists?

Would I be correct in guessing that pages which are on the
per-rss-container lists are also eligible for reclaim off the traditional
page LRUs?  If so, how does that work?  When a page gets freed off the
per-zone LRUs does it also get removed from the per-rss_container LRU?  But
how can this be right?  Pages can get taken off the LRU and freed at
interrupt time, and this code isn't interrupt-safe.

I note that this lock is not irq-safe, whereas the lru locks are irq-safe.
So we don't perform the rotate_reclaimable_page() operation within the RSS
container?  I think we could do so.  I wonder if this was considered.

A description of how all this code works would help a lot.

> +static unsigned long isolate_container_pages(unsigned long nr_to_scan,
> +  struct list_head *src, struct list_head *dst,
> +  unsigned long *scanned, struct zone *zone, int mode)
> +{
> + unsigned long nr_taken = 0;
> + struct page *page;
> + struct page_container *pc;
> + unsigned long scan;
> + LIST_HEAD(pc_list);
> +
> + for (scan = 0; scan < nr_to_scan && !list_empty(src); scan++) {
> +  pc = list_entry(src->prev, struct page_container, list);
> +  page = pc->page;
> +  if (page_zone(page) != zone)
> +   continue;

That page_zone() check is interesting.  What's going on here?

I'm suspecting that we have a problem here: if there are a lot of pages on
*src which are in the wrong zone, we can suffer reclaim distress leading to
omm-killings, or excessive CPU consumption?

```
> +  list_move(&pc->list, &pc_list);
> +
> +  if (__isolate_lru_page(page, mode) == 0) {
> +   list_move(&page->lru, dst);
> +   nr_taken++;
> +  }
> + }
> +
> + list_splice(&pc_list, src);
> +
> + *scanned = scan;
> + return nr_taken;
> +}
> +
> +unsigned long isolate_pages_in_container(unsigned long nr_to_scan,
> +  struct list_head *dst, unsigned long *scanned,
> +  int order, int mode, struct zone *zone,
> +  struct rss_container *rss, int active)
> +{
> + unsigned long ret;
> +
> + spin_lock(&rss->res.lock);
> + if (active)
> +  ret = isolate_container_pages(nr_to_scan, &rss->active_list,
> +    dst, scanned, zone, mode);
> + else
> +  ret = isolate_container_pages(nr_to_scan, &rss->inactive_list,
> +    dst, scanned, zone, mode);
> + spin_unlock(&rss->res.lock);
> + return ret;
> +}
> +
>  void container_rss_add(struct page_container *pc)
>  {
>   struct page *pg;
>
> ...
>
> +#ifdef CONFIG_RSS_CONTAINER
> +unsigned long try_to_free_pages_in_container(struct rss_container *cnt)
> +{
> + struct scan_control sc = {
> +  .gfp_mask = GFP_KERNEL,
```

> +  .may_writepage = 1,
> +  .swap_cluster_max = 1,
> +  .may_swap = 1,
> +  .swappiness = vm_swappiness,
> +  .order = 0, /* in this case we wanted one page only */
> +  .cnt = cnt,
> +  .isolate_pages = isolate_pages_in_container,
> + };
> + int node;
> + struct zone **zones;
> +
> + for_each_online_node(node) {
> +#ifdef CONFIG_HIGHMEM
> +  zones = NODE_DATA(node)->node_zonelists[ZONE_HIGHMEM].zones;
> +  if (do_try_to_free_pages(zones, sc.gfp_mask, &sc))
> +   return 1;
> +#endif
> +  zones = NODE_DATA(node)->node_zonelists[ZONE_NORMAL].zones;
> +  if (do_try_to_free_pages(zones, sc.gfp_mask, &sc))
> +   return 1;
> + }

Definitely need to handle ZONE_DMA32 and ZONE_DMA (some architectures put
all memory into ZONE_DMA (or they used to))

> + return 0;
> +}
> +#endif
> +
>  unsigned long try_to_free_pages(struct zone **zones, int order, gfp_t gfp_mask)
>  {
>   struct scan_control sc = {

---

## Subject: Re: [PATCH 8/8] Per-container pages reclamation
Posted by Vaidyanathan Srinivas on Thu, 31 May 2007 08:22:13 GMT
View Forum Message <> Reply to Message

Andrew Morton wrote:
> On Wed, 30 May 2007 19:42:26 +0400
> Pavel Emelianov <xemul@openvz.org> wrote:
>
[snip]
>>
>> +#ifdef CONFIG_RSS_CONTAINER
>> +unsigned long try_to_free_pages_in_container(struct rss_container *cnt)
>> +{
>> + struct scan_control sc = {

```
>> +  .gfp_mask = GFP_KERNEL,
>> +  .may_writepage = 1,
>> +  .swap_cluster_max = 1,
>> +  .may_swap = 1,
>> +  .swappiness = vm_swappiness,
>> +  .order = 0, /* in this case we wanted one page only */
>> +  .cnt = cnt,
>> +  .isolate_pages = isolate_pages_in_container,
>> + };
>> + int node;
>> + struct zone **zones;
>> +
>> + for_each_online_node(node) {
>> +#ifdef CONFIG_HIGHMEM
>> +  zones = NODE_DATA(node)->node_zonelists[ZONE_HIGHMEM].zones;
>> +  if (do_try_to_free_pages(zones, sc.gfp_mask, &sc))
>> +   return 1;
>> +#endif
>> +  zones = NODE_DATA(node)->node_zonelists[ZONE_NORMAL].zones;
>> +  if (do_try_to_free_pages(zones, sc.gfp_mask, &sc))
>> +   return 1;
>> + }
>
> Definitely need to handle ZONE_DMA32 and ZONE_DMA (some architectures put
> all memory into ZONE_DMA (or they used to))

Can we iterate from MAX_NR_ZONES-1 to automatically choose the right
zone for the system?

for_each_online_node(node)
 for (i = MAX_NR_ZONES - 1; i >= 0; i--) {
   zones = NODE_DATA(node)->node_zonelists[i].zones;
  if (do_try_to_free_pages(zones, sc.gfp_mask, &sc))
   return 1;

The above code works on my PPC64 box where all memory is in ZOME_DMA.

>> + return 0;
>> +}
>> +#endif
>> +
>>  unsigned long try_to_free_pages(struct zone **zones, int order, gfp_t gfp_mask)
>>  {
>>   struct scan_control sc = {
> -
> To unsubscribe from this list: send the line "unsubscribe linux-kernel" in
> the body of a message to majordomo@vger.kernel.org
> More majordomo info at  http://vger.kernel.org/majordomo-info.html
```

> Please read the FAQ at  http://www.tux.org/lkml/
>

---

Subject: Re: [PATCH 4/8] RSS container core
Posted by Pavel Emelianov on Thu, 31 May 2007 08:57:26 GMT
View Forum Message <> Reply to Message

Andrew Morton wrote:
>> +#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 */
>>
> eh?  What is gcc's problem here?

When this line is missed and CONFIG_RSS_COUNTER is n the following
warnings are produced:

  CC      mm/memory.o

>> + return 0;
>> +}
>>
>> ...
>>
>> +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;
>

> If we fail partway through here, do the thus-far-created fiels get cleaned up?

Yes. As far as I see from Paul's code when one of the files is failed to
be created the whole container is cleaned up.

>> + 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,
>> +};
>
> Did this need kernel-wide scope?

Yes. In include/linux/countainer_subsys.h we declared the SUBSYS(rss)
that is expanded into "extern struct container_subsys rss_subsys;" in the
kernel/container.c file. Further the pointer to it is injected into the array
of subsystems (static struct container_subsys *subsys[]; at the top of the file).

---

Subject: Re: [PATCH 8/8] Per-container pages reclamation
Posted by Balbir Singh on Thu, 31 May 2007 09:22:31 GMT
View Forum Message <> Reply to Message

Vaidyanathan Srinivasan wrote:
>
> Andrew Morton wrote:
>> On Wed, 30 May 2007 19:42:26 +0400
>> Pavel Emelianov <xemul@openvz.org> wrote:
>>
> [snip]
>>> +#ifdef CONFIG_RSS_CONTAINER
>>> +unsigned long try_to_free_pages_in_container(struct rss_container *cnt)
>>> +{
>>> + struct scan_control sc = {
>>> + .gfp_mask = GFP_KERNEL,
>>> + .may_writepage = 1,
>>> + .swap_cluster_max = 1,
>>> + .may_swap = 1,
>>> + .swappiness = vm_swappiness,
>>> + .order = 0, /* in this case we wanted one page only */
>>> + .cnt = cnt,

>>> +  .isolate_pages = isolate_pages_in_container,
>>> + };
>>> + int node;
>>> + struct zone **zones;
>>> +
>>> + for_each_online_node(node) {
>>> +#ifdef CONFIG_HIGHMEM
>>> +  zones = NODE_DATA(node)->node_zonelists[ZONE_HIGHMEM].zones;
>>> +  if (do_try_to_free_pages(zones, sc.gfp_mask, &sc))
>>> +   return 1;
>>> +#endif
>>> +  zones = NODE_DATA(node)->node_zonelists[ZONE_NORMAL].zones;
>>> +  if (do_try_to_free_pages(zones, sc.gfp_mask, &sc))
>>> +   return 1;
>>> + }
>> Definitely need to handle ZONE_DMA32 and ZONE_DMA (some architectures put
>> all memory into ZONE_DMA (or they used to))
>

Won't the node_zonelists[ZONE_NORMAL].zones contain all lower zones (including
ZONE_DMA and ZONE_DMA32) from various nodes organized by cost?

I need to go lookup the node_zonelists creation.

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

---

Subject: Re: [PATCH 8/8] Per-container pages reclamation
Posted by Pavel Emelianov on Thu, 31 May 2007 10:31:21 GMT
View Forum Message <> Reply to Message

Andrew Morton wrote:
> On Wed, 30 May 2007 19:42:26 +0400
> Pavel Emelianov <xemul@openvz.org> wrote:
>
>> Implement try_to_free_pages_in_container() to free the
>> pages in container that has run out of memory.
>>
>> The scan_control->isolate_pages() function is set to
>> isolate_pages_in_container() that isolates the container
>> pages only. The exported __isolate_lru_page() call
>> makes things look simpler than in the previous version.
>>

>> Includes fix from Balbir Singh <balbir@in.ibm.com>
>>
>>  }
>>
>> +void container_rss_move_lists(struct page *pg, bool active)
>> +{
>> + struct rss_container *rss;
>> + struct page_container *pc;
>> +
>> + if (!page_mapped(pg))
>> +  return;
>> +
>> + pc = page_container(pg);
>> + if (pc == NULL)
>> +  return;
>> +
>> + rss = pc->cnt;
>> +
>> + spin_lock(&rss->res.lock);
>> + if (active)
>> +  list_move(&pc->list, &rss->active_list);
>> + else
>> +  list_move(&pc->list, &rss->inactive_list);
>> + spin_unlock(&rss->res.lock);
>> +}
>
> This is an interesting-looking function.  Please document it?
>
> I'm inferring that the rss container has an active and inactive list and
> that this basically follows the same operation as the traditional per-zone
> lists?

Yes - each container tries to look like a zone, i.e. have two lists
and hold the pages there in LRU manner. The problem here (and it was
pointed out by you lower) is that these two lists store pages from
all the zones. This is still the largest TODO in this patchset. Sorry
for not pointing this out explicitly.

> Would I be correct in guessing that pages which are on the
> per-rss-container lists are also eligible for reclaim off the traditional
> page LRUs?  If so, how does that work?  When a page gets freed off the

Yes. All the pages are accessible from booth - global and per-container
LRU lists and reclamation can be performed from booth.

> per-zone LRUs does it also get removed from the per-rss_container LRU?  But
> how can this be right?

I don't get your idea here.

> Pages can get taken off the LRU and freed at
> interrupt time, and this code isn't interrupt-safe.

Actually, all the places we move the page within the container lists
are not in interrupts. But I have found at least one when the page
is being moved from one list to another, this place is IRQ-accessible,
but we don't move the page in container. Thus we have a BUG in
LRU maintenance but not in interrupt-safeness.

I will recheck this.

> I note that this lock is not irq-safe, whereas the lru locks are irq-safe.
> So we don't perform the rotate_reclaimable_page() operation within the RSS
> container?  I think we could do so.  I wonder if this was considered.
>
> A description of how all this code works would help a lot.
>
>> +static unsigned long isolate_container_pages(unsigned long nr_to_scan,
>> +  struct list_head *src, struct list_head *dst,
>> +  unsigned long *scanned, struct zone *zone, int mode)
>> +{
>> + unsigned long nr_taken = 0;
>> + struct page *page;
>> + struct page_container *pc;
>> + unsigned long scan;
>> + LIST_HEAD(pc_list);
>> +
>> + for (scan = 0; scan < nr_to_scan && !list_empty(src); scan++) {
>> +  pc = list_entry(src->prev, struct page_container, list);
>> +  page = pc->page;
>> +  if (page_zone(page) != zone)
>> +   continue;
>
> That page_zone() check is interesting.  What's going on here?
>
> I'm suspecting that we have a problem here: if there are a lot of pages on
> *src which are in the wrong zone, we can suffer reclaim distress leading to
> omm-killings, or excessive CPU consumption?

That's the TODO I have told above.

---

Subject: Re: [PATCH 8/8] Per-container pages reclamation
Posted by Andrew Morton on Thu, 31 May 2007 17:58:42 GMT
View Forum Message <> Reply to Message

On Thu, 31 May 2007 14:35:43 +0400
Pavel Emelianov <xemul@openvz.org> wrote:

> > Would I be correct in guessing that pages which are on the
> > per-rss-container lists are also eligible for reclaim off the traditional
> > page LRUs?  If so, how does that work?  When a page gets freed off the
>
> Yes. All the pages are accessible from booth - global and per-container
> LRU lists and reclamation can be performed from booth.
>
> > per-zone LRUs does it also get removed from the per-rss_container LRU?  But
> > how can this be right?
>
> I don't get your idea here.

If we have a page which is on the zone LRU with refcount=1 and someone does
put_page() on it, we will take that page off the zone LRU and then actually
free the page.

I am assuming that your patches change that logic so that we will also
remove that page from the per-container LRU at the same time?

If so, the problem which I see is that, under rare circumstances, that
final put_page() will occur from interrupt context.  Hence we would be
trying to remove the page from the per-container LRU at interrupt time.
But the locks which you have in there are not suited for taking from
interrupt context.

---

Subject: Re: [PATCH 5/8] RSS accounting hooks over the code
Posted by Balbir Singh on Fri, 01 Jun 2007 06:48:10 GMT
View Forum Message <> Reply to Message

Andrew Morton wrote:
> On Wed, 30 May 2007 19:34:18 +0400
> Pavel Emelianov <xemul@openvz.org> wrote:
>
>> As described above, pages are charged to their first touchers.
>> The first toucher is determined using pages' _mapcount
>> manipulations in rmap calls.
>>
>> Page is charged in two stages:
>> 1. preparation, in which the resource availability is checked.
>>    This stage may lead to page reclamation, thus it is performed
>>    in a "might-sleep" places;
>> 2. the container assignment to page. This is done in an atomic
>>    code that handles races between multiple touchers.
>

> I suppose we need to think about what to do about higher-order pages, and
> compound pages, and hugetlb memory.

Yes, that needs to be done eventually. I feel that support can be added
incrementally once we have a stable controller.

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

---

Subject: Re: [PATCH 7/8] Scanner changes needed to implement per-container scanner
Posted by Balbir Singh on Fri, 01 Jun 2007 06:50:03 GMT
View Forum Message <> Reply to Message

Andrew Morton wrote:
>
>> + nr_reclaimed += shrink_zones(priority, zones, sc);
>> + if (sc->cnt == NULL)
>> + shrink_slab(sc->nr_scanned, gfp_mask, lru_pages);
>
> We don't we shrink slab if called to shrink a container.
>
> This is a fundamental design decision, and a design shortcoming. A full
> discussion of this is absolutely appropriate to the patch changelog.
> Please don't just hide stuff like this in the patch and leave people
> wondering, or ignorant.

Yes, we don't because we do not account for slab usage right now. We account
only for memory allocated to user space. A good fat comment will help here.

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

---

Subject: Re: [PATCH 8/8] Per-container pages reclamation
Posted by Balbir Singh on Fri, 01 Jun 2007 07:02:33 GMT
View Forum Message <> Reply to Message

Andrew Morton wrote:

> On Wed, 30 May 2007 19:42:26 +0400
> Pavel Emelianov <xemul@openvz.org> wrote:
>
>> Implement try_to_free_pages_in_container() to free the
>> pages in container that has run out of memory.
>>
>> The scan_control->isolate_pages() function is set to
>> isolate_pages_in_container() that isolates the container
>> pages only. The exported __isolate_lru_page() call
>> makes things look simpler than in the previous version.
>>
>> Includes fix from Balbir Singh <balbir@in.ibm.com>
>>
>>  }
>>
>> +void container_rss_move_lists(struct page *pg, bool active)
>> +{
>> + struct rss_container *rss;
>> + struct page_container *pc;
>> +
>> + if (!page_mapped(pg))
>> +  return;
>> +
>> + pc = page_container(pg);
>> + if (pc == NULL)
>> +  return;
>> +
>> + rss = pc->cnt;
>> +
>> + spin_lock(&rss->res.lock);
>> + if (active)
>> +  list_move(&pc->list, &rss->active_list);
>> + else
>> +  list_move(&pc->list, &rss->inactive_list);
>> + spin_unlock(&rss->res.lock);
>> +}
>
> This is an interesting-looking function.  Please document it?
>

Will do. This function is called when we want to move a page in
the LRU list. This could happen when a page is activated or when
reclaim finds that a particular page cannot be reclaimed right now.

> I'm inferring that the rss container has an active and inactive list and
> that this basically follows the same operation as the traditional per-zone
> lists?
>

Yes, correct.

> Would I be correct in guessing that pages which are on the
> per-rss-container lists are also eligible for reclaim off the traditional
> page LRUs?  If so, how does that work?  When a page gets freed off the
> per-zone LRUs does it also get removed from the per-rss_container LRU?  But
> how can this be right?  Pages can get taken off the LRU and freed at
> interrupt time, and this code isn't interrupt-safe.
>

Yes, before a page is reclaimed from the global LRU list, we go through
page_remove_rmap() in try_to_unmap(). Pages are removed from the container
LRU before they are reclaimed.

> I note that this lock is not irq-safe, whereas the lru locks are irq-safe.
> So we don't perform the rotate_reclaimable_page() operation within the RSS
> container?  I think we could do so.  I wonder if this was considered.
>

The lock needs to be interrupt safe.

> A description of how all this code works would help a lot.
>
>> +static unsigned long isolate_container_pages(unsigned long nr_to_scan,
>> +  struct list_head *src, struct list_head *dst,
>> +  unsigned long *scanned, struct zone *zone, int mode)
>> +{
>> + unsigned long nr_taken = 0;
>> + struct page *page;
>> + struct page_container *pc;
>> + unsigned long scan;
>> + LIST_HEAD(pc_list);
>> +
>> + for (scan = 0; scan < nr_to_scan && !list_empty(src); scan++) {
>> +  pc = list_entry(src->prev, struct page_container, list);
>> +  page = pc->page;
>> +  if (page_zone(page) != zone)
>> +   continue;
>
> That page_zone() check is interesting.  What's going on here?
>
> I'm suspecting that we have a problem here: if there are a lot of pages on
> *src which are in the wrong zone, we can suffer reclaim distress leading to
> omm-killings, or excessive CPU consumption?
>

We discussed this on lkml. Basically, now for every zone we try to reclaim

pages from the container, it increases CPU utilization if we choose the wrong
zone to reclaim from, but it provides the following benefit

Code reuse (shrink_zone* is reused along with helper functions). I am not sure
if Pavel had any other benefits in mind like benefits on a NUMA box.

>> + for_each_online_node(node) {
>> +#ifdef CONFIG_HIGHMEM
>> +  zones = NODE_DATA(node)->node_zonelists[ZONE_HIGHMEM].zones;
>> +  if (do_try_to_free_pages(zones, sc.gfp_mask, &sc))
>> +   return 1;
>> +#endif
>> +  zones = NODE_DATA(node)->node_zonelists[ZONE_NORMAL].zones;
>> +  if (do_try_to_free_pages(zones, sc.gfp_mask, &sc))
>> +   return 1;
>> + }
>
> Definitely need to handle ZONE_DMA32 and ZONE_DMA (some architectures put
> all memory into ZONE_DMA (or they used to))
>

node_zonelists[ZONE_NORMAL].zones should contain ZONE_DMA and ZONE_DMA32 right?

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

---

Subject: Re: [PATCH 7/8] Scanner changes needed to implement per-container scanner
Posted by Pavel Emelianov on Fri, 01 Jun 2007 07:36:09 GMT
View Forum Message <> Reply to Message

Balbir Singh wrote:
> Andrew Morton wrote:
>>> +  nr_reclaimed += shrink_zones(priority, zones, sc);
>>> +  if (sc->cnt == NULL)
>>> +   shrink_slab(sc->nr_scanned, gfp_mask, lru_pages);
>> We don't we shrink slab if called to shrink a container.
>>
>> This is a fundamental design decision, and a design shortcoming.  A full
>> discussion of this is absolutely appropriate to the patch changelog.
>> Please don't just hide stuff like this in the patch and leave people
>> wondering, or ignorant.

>
> Yes, we don't because we do not account for slab usage right now. We account
> only for memory allocated to user space. A good fat comment will help here.
>
>

I have already added the comment. But the problem is not in that we
do not account for kernel memory. Shrinking slabs won't unmap any
pages from user-space and thus won't help user to charge more. This
will only make kernel suffer from re-creation of objects.

Thanks,
Pavel.

---

Subject: Re: [PATCH 7/8] Scanner changes needed to implement per-container scanner
Posted by Balbir Singh on Fri, 01 Jun 2007 07:38:03 GMT
View Forum Message <> Reply to Message

Pavel Emelianov wrote:
> Balbir Singh wrote:
>> Andrew Morton wrote:
>>>> +  nr_reclaimed += shrink_zones(priority, zones, sc);
>>>> +  if (sc->cnt == NULL)
>>>> +   shrink_slab(sc->nr_scanned, gfp_mask, lru_pages);
>>> We don't we shrink slab if called to shrink a container.
>>>
>>> This is a fundamental design decision, and a design shortcoming.  A full
>>> discussion of this is absolutely appropriate to the patch changelog.
>>> Please don't just hide stuff like this in the patch and leave people
>>> wondering, or ignorant.
>> Yes, we don't because we do not account for slab usage right now. We account
>> only for memory allocated to user space. A good fat comment will help here.
>>
>>
>
> I have already added the comment. But the problem is not in that we
> do not account for kernel memory. Shrinking slabs won't unmap any
> pages from user-space and thus won't help user to charge more. This
> will only make kernel suffer from re-creation of objects.

I meant the same thing. Thanks for adding the comment.

--
Warm Regards,
Balbir Singh
Linux Technology Center

Subject: Re: [PATCH 8/8] Per-container pages reclamation
Posted by Pavel Emelianov on Fri, 01 Jun 2007 07:40:12 GMT
View Forum Message <> Reply to Message

Andrew Morton wrote:
> On Thu, 31 May 2007 14:35:43 +0400
> Pavel Emelianov <xemul@openvz.org> wrote:
>
>>> Would I be correct in guessing that pages which are on the
>>> per-rss-container lists are also eligible for reclaim off the traditional
>>> page LRUs? If so, how does that work? When a page gets freed off the
>> Yes. All the pages are accessible from booth - global and per-container
>> LRU lists and reclamation can be performed from booth.
>>
>>> per-zone LRUs does it also get removed from the per-rss_container LRU? But
>>> how can this be right?
>> I don't get your idea here.
>
> If we have a page which is on the zone LRU with refcount=1 and someone does
> put_page() on it, we will take that page off the zone LRU and then actually
> free the page.
>
> I am assuming that your patches change that logic so that we will also
> remove that page from the per-container LRU at the same time?

Oh, I see. No that will work another way. Page stays in per-container
LRU lists as long as it is mapped. When the last process is unmapping
the page, it it is removed, but stays in global LRU till its refcount
drops to 0.

Since all unmaps happen from process-context this looks to be safe.

> If so, the problem which I see is that, under rare circumstances, that
> final put_page() will occur from interrupt context. Hence we would be
> trying to remove the page from the per-container LRU at interrupt time.
> But the locks which you have in there are not suited for taking from
> interrupt context.
>
>

Thanks,
Pavel.

Subject: Re: [PATCH 8/8] Per-container pages reclamation
Posted by Pavel Emelianov on Fri, 01 Jun 2007 07:47:18 GMT
View Forum Message <> Reply to Message

Balbir Singh wrote:
> Andrew Morton wrote:
>> On Wed, 30 May 2007 19:42:26 +0400
>> Pavel Emelianov <xemul@openvz.org> wrote:
>>
>>> Implement try_to_free_pages_in_container() to free the
>>> pages in container that has run out of memory.
>>>
>>> The scan_control->isolate_pages() function is set to
>>> isolate_pages_in_container() that isolates the container
>>> pages only. The exported __isolate_lru_page() call
>>> makes things look simpler than in the previous version.
>>>
>>> Includes fix from Balbir Singh <balbir@in.ibm.com>
>>>
>>>  }
>>>
>>> +void container_rss_move_lists(struct page *pg, bool active)
>>> +{
>>> + struct rss_container *rss;
>>> + struct page_container *pc;
>>> +
>>> + if (!page_mapped(pg))
>>> +  return;
>>> +
>>> + pc = page_container(pg);
>>> + if (pc == NULL)
>>> +  return;
>>> +
>>> + rss = pc->cnt;
>>> +
>>> + spin_lock(&rss->res.lock);
>>> + if (active)
>>> +  list_move(&pc->list, &rss->active_list);
>>> + else
>>> +  list_move(&pc->list, &rss->inactive_list);
>>> + spin_unlock(&rss->res.lock);
>>> +}
>> This is an interesting-looking function.  Please document it?
>>
>
> Will do. This function is called when we want to move a page in
> the LRU list. This could happen when a page is activated or when
> reclaim finds that a particular page cannot be reclaimed right now.
>

>> I'm inferring that the rss container has an active and inactive list and
>> that this basically follows the same operation as the traditional per-zone
>> lists?
>>
>
> Yes, correct.
>
>> Would I be correct in guessing that pages which are on the
>> per-rss-container lists are also eligible for reclaim off the traditional
>> page LRUs?  If so, how does that work?  When a page gets freed off the
>> per-zone LRUs does it also get removed from the per-rss_container LRU?  But
>> how can this be right?  Pages can get taken off the LRU and freed at
>> interrupt time, and this code isn't interrupt-safe.
>>
>
> Yes, before a page is reclaimed from the global LRU list, we go through
> page_remove_rmap() in try_to_unmap(). Pages are removed from the container
> LRU before they are reclaimed.

Yup. And this makes possible for additional charge.

>> I note that this lock is not irq-safe, whereas the lru locks are irq-safe.
>> So we don't perform the rotate_reclaimable_page() operation within the RSS
>> container?  I think we could do so.  I wonder if this was considered.
>>
>
> The lock needs to be interrupt safe.

Agree. I have found one place where we lost moving the page across
lists and this place is interrupt-accessible.

>> A description of how all this code works would help a lot.
>>
>>> +static unsigned long isolate_container_pages(unsigned long nr_to_scan,
>>> +  struct list_head *src, struct list_head *dst,
>>> +  unsigned long *scanned, struct zone *zone, int mode)
>>> +{
>>> + unsigned long nr_taken = 0;
>>> + struct page *page;
>>> + struct page_container *pc;
>>> + unsigned long scan;
>>> + LIST_HEAD(pc_list);
>>> +
>>> + for (scan = 0; scan < nr_to_scan && !list_empty(src); scan++) {
>>> +  pc = list_entry(src->prev, struct page_container, list);
>>> +  page = pc->page;
>>> +  if (page_zone(page) != zone)
>>> +   continue;

>> That page_zone() check is interesting.  What's going on here?
>>
>> I'm suspecting that we have a problem here: if there are a lot of pages on
>> *src which are in the wrong zone, we can suffer reclaim distress leading to
>> omm-killings, or excessive CPU consumption?
>>
>
> We discussed this on lkml. Basically, now for every zone we try to reclaim
> pages from the container, it increases CPU utilization if we choose the wrong
> zone to reclaim from, but it provides the following benefit
>
> Code reuse (shrink_zone* is reused along with helper functions). I am not sure
> if Pavel had any other benefits in mind like benefits on a NUMA box.

Actually I planned to make RSS container look like a standalone machine
with active/inactive lists being per-zone/per-node. This will take all the
benefits from the current scanner.

Right now the lists are "global", but I'm looking at how zones are organized
to find out how to split them.

>>> + for_each_online_node(node) {
>>> +#ifdef CONFIG_HIGHMEM
>>> +  zones = NODE_DATA(node)->node_zonelists[ZONE_HIGHMEM].zones;
>>> +  if (do_try_to_free_pages(zones, sc.gfp_mask, &sc))
>>> +   return 1;
>>> +#endif
>>> +  zones = NODE_DATA(node)->node_zonelists[ZONE_NORMAL].zones;
>>> +  if (do_try_to_free_pages(zones, sc.gfp_mask, &sc))
>>> +   return 1;
>>> + }
>> Definitely need to handle ZONE_DMA32 and ZONE_DMA (some architectures put
>> all memory into ZONE_DMA (or they used to))
>>
>
> node_zonelists[ZONE_NORMAL].zones should contain ZONE_DMA and ZONE_DMA32 right?

Wait. Do you mean, that zones intersect with each other???

---

Subject: Re: [PATCH 8/8] Per-container pages reclamation
Posted by Andrew Morton on Fri, 01 Jun 2007 07:49:22 GMT
View Forum Message <> Reply to Message

On Fri, 01 Jun 2007 11:44:25 +0400 Pavel Emelianov <xemul@openvz.org> wrote:

> Andrew Morton wrote:
> > On Thu, 31 May 2007 14:35:43 +0400

> > Pavel Emelianov <xemul@openvz.org> wrote:
> >
> >>> Would I be correct in guessing that pages which are on the
> >>> per-rss-container lists are also eligible for reclaim off the traditional
> >>> page LRUs?  If so, how does that work?  When a page gets freed off the
> >> Yes. All the pages are accessible from booth - global and per-container
> >> LRU lists and reclamation can be performed from booth.
> >>
> >>> per-zone LRUs does it also get removed from the per-rss_container LRU?  But
> >>> how can this be right?
> >> I don't get your idea here.
> >
> > If we have a page which is on the zone LRU with refcount=1 and someone does
> > put_page() on it, we will take that page off the zone LRU and then actually
> > free the page.
> >
> > I am assuming that your patches change that logic so that we will also
> > remove that page from the per-container LRU at the same time?
>
> Oh, I see. No that will work another way. Page stays in per-container
> LRU lists as long as it is mapped. When the last process is unmapping
> the page, it it is removed, but stays in global LRU till its refcount
> drops to 0.

ho hum, OK.  This wasn't obvious from the code and it isn't something
I should have learned by emailing you guys!  Please have a think about
preparing an overall decription of the design and implementation of this
rather important kernel change?

---

Subject: Re: [PATCH 8/8] Per-container pages reclamation
Posted by Pavel Emelianov on Fri, 01 Jun 2007 09:23:10 GMT
View Forum Message <> Reply to Message

Balbir Singh wrote:

[snip]

>>>> + for_each_online_node(node) {
>>>> +#ifdef CONFIG_HIGHMEM
>>>> +  zones = NODE_DATA(node)->node_zonelists[ZONE_HIGHMEM].zones;
>>>> +  if (do_try_to_free_pages(zones, sc.gfp_mask, &sc))
>>>> +   return 1;
>>>> +#endif
>>>> +  zones = NODE_DATA(node)->node_zonelists[ZONE_NORMAL].zones;
>>>> +  if (do_try_to_free_pages(zones, sc.gfp_mask, &sc))
>>>> +   return 1;
>>>> + }

>>> Definitely need to handle ZONE_DMA32 and ZONE_DMA (some architectures put
>>> all memory into ZONE_DMA (or they used to))
>
> Won't the node_zonelists[ZONE_NORMAL].zones contain all lower zones (including
> ZONE_DMA and ZONE_DMA32) from various nodes organized by cost?
>
> I need to go lookup the node_zonelists creation.

OK, I've got it. The node's zonelist indexed with ZONE_XXX contains
the zone specified and all the zones beyond it. Thus shrinking from
ZONE_HIGHMEM (ZONE_NORMAL if the latter does not exist) is enough.

Thanks,
Pavel

---

Subject: Re: [PATCH 8/8] Per-container pages reclamation
Posted by Balbir Singh on Fri, 01 Jun 2007 09:23:30 GMT
View Forum Message <> Reply to Message

Pavel Emelianov wrote:
> Balbir Singh wrote:
>
> [snip]
>
>>>>> + for_each_online_node(node) {
>>>>> +#ifdef CONFIG_HIGHMEM
>>>>> +  zones = NODE_DATA(node)->node_zonelists[ZONE_HIGHMEM].zones;
>>>>> +  if (do_try_to_free_pages(zones, sc.gfp_mask, &sc))
>>>>> +   return 1;
>>>>> +#endif
>>>>> +  zones = NODE_DATA(node)->node_zonelists[ZONE_NORMAL].zones;
>>>>> +  if (do_try_to_free_pages(zones, sc.gfp_mask, &sc))
>>>>> +   return 1;
>>>>> + }
>>>> Definitely need to handle ZONE_DMA32 and ZONE_DMA (some architectures put
>>>> all memory into ZONE_DMA (or they used to))
>> Won't the node_zonelists[ZONE_NORMAL].zones contain all lower zones (including
>> ZONE_DMA and ZONE_DMA32) from various nodes organized by cost?
>>
>> I need to go lookup the node_zonelists creation.
>
>
> OK, I've got it. The node's zonelist indexed with ZONE_XXX contains
> the zone specified and all the zones beyond it. Thus shrinking from
> ZONE_HIGHMEM (ZONE_NORMAL if the latter does not exist) is enough.
>

> Thanks,
> Pavel

Exactly!

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

---