Subject: [RFC][PATCH][0/4] Memory controller (RSS Control)
Posted by Balbir Singh on Mon, 19 Feb 2007 06:50:19 GMT
View Forum Message <> Reply to Message

This patch applies on top of Paul Menage's container patches (V7) posted at

http://lkml.org/lkml/2007/2/12/88

It implements a controller within the containers framework for limiting
memory usage (RSS usage).

The memory controller was discussed at length in the RFC posted to lkml
http://lkml.org/lkml/2006/10/30/51

Steps to use the controller
---

0. Download the patches, apply the patches
1. Turn on CONFIG_CONTAINER_MEMCTLR in kernel config, build the kernel
   and boot into the new kernel
2. mount -t container container -o memctlr /<mount point>
3. cd /<mount point>
   optionally do (mkdir <directory>; cd <directory>) under /<mount point>
4. echo $$ > tasks (attaches the current shell to the container)
5. echo -n (limit value) > memctlr_limit
6. cat memctlr_usage
7. Run tasks, check the usage of the controller, reclaim behaviour
8. Report bugs, get bug fixes and iterate (goto step 0).

Advantages of the patchset
---

1. Zero overhead in struct page (struct page is not expanded)
2. Minimal changes to the core-mm code
3. Shared pages are not reclaimed unless all mappings belong to overlimit
   containers.
4. It can be used to debug drivers/applications/kernel components in a
   constrained memory environment (similar to mem=XXX option), except that
   several containers can be created simultaneously without rebooting and
   the limits can be changed. NOTE: There is no support for limiting
   kernel memory allocations and page cache control (presently).

Testing
---
Ran kernbench and lmbench with containers enabled (container filesystem not
mounted), they seemed to run fine
Created containers, attached tasks to containers with lower limits than

the memory the tasks require (memory hog tests) and ran some basic tests on them

TODO's and improvement areas
---
1. Come up with cool page replacement algorithms for containers
   (if possible without any changes to struct page)
2. Add page cache control
3. Add kernel memory allocator control
4. Extract benchmark numbers and overhead data

Comments & criticism are welcome.

Series
---
memctlr-setup.patch
memctlr-acct.patch
memctlr-reclaim-on-limit.patch
memctlr-doc.patch

--
Warm Regards,
Balbir Singh

---

Subject: [RFC][PATCH][1/4] RSS controller setup
Posted by Balbir Singh on Mon, 19 Feb 2007 06:50:26 GMT
View Forum Message <> Reply to Message

This patch sets up the basic controller infrastructure on top of the containers infrastructure. Two files are provided for monitoring and control memctlr_usage and memctlr_limit.

memctlr_usage shows the current usage (in pages, of RSS) and the limit set by the user.

memctlr_limit can be used to set a limit on the RSS usage of the resource. A special value of 0, indicates that the usage is unlimited. The limit is set in units of pages.

Signed-off-by: <balbir@in.ibm.com>
---

```
 include/linux/memctlr.h |   22 ++++++
 init/Kconfig            |    7 +
 mm/Makefile             |    1
 mm/memctlr.c            |  169 ++++++++++++++++++++++++++++++++++++++++++++++++++
```

```
 4 files changed, 199 insertions(+)

diff -puN /dev/null include/linux/memctlr.h
--- /dev/null 2007-02-02 22:51:23.000000000 +0530
+++ linux-2.6.20-balbir/include/linux/memctlr.h 2007-02-16 00:22:11.000000000 +0530
@@ -0,0 +1,22 @@
+/* memctlr.h - Memory Controller for containers
+ *
+ * Copyright (C) Balbir Singh,   IBM Corp. 2006-2007
+ *
+ * This program is free software; you can redistribute it and/or modify it
+ * under the terms of version 2.1 of the GNU Lesser General Public License
+ * as published by the Free Software Foundation.
+ *
+ * This program is distributed in the hope that it would be useful, but
+ * WITHOUT ANY WARRANTY; without even the implied warranty of
+ * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
+ */
+
+#ifndef _LINUX_MEMCTLR_H
+#define _LINUX_MEMCTLR_H
+
+#ifdef CONFIG_CONTAINER_MEMCTLR
+
+#else /* CONFIG_CONTAINER_MEMCTLR  */
+
+#endif /* CONFIG_CONTAINER_MEMCTLR */
+#endif /* _LINUX_MEMCTLR_H */
diff -puN init/Kconfig~memctlr-setup init/Kconfig
--- linux-2.6.20/init/Kconfig~memctlr-setup 2007-02-15 21:58:42.000000000 +0530
+++ linux-2.6.20-balbir/init/Kconfig 2007-02-15 21:58:42.000000000 +0530
@@ -306,6 +306,13 @@ config CONTAINER_NS
          for instance virtual servers and checkpoint/restart
          jobs.

+config CONTAINER_MEMCTLR
+ bool "A simple RSS based memory controller"
+ select CONTAINERS
+ help
+   Provides a simple Resource Controller for monitoring and
+   controlling the total Resident Set Size of the tasks in a container
+
 config RELAY
  bool "Kernel->user space relay support (formerly relayfs)"
  help
diff -puN mm/Makefile~memctlr-setup mm/Makefile
--- linux-2.6.20/mm/Makefile~memctlr-setup 2007-02-15 21:58:42.000000000 +0530
+++ linux-2.6.20-balbir/mm/Makefile 2007-02-15 21:58:42.000000000 +0530
```

```
@@ -29,3 +29,4 @@ obj-$(CONFIG_MEMORY_HOTPLUG) += memory_h
 obj-$(CONFIG_FS_XIP) += filemap_xip.o
 obj-$(CONFIG_MIGRATION) += migrate.o
 obj-$(CONFIG_SMP) += allocpercpu.o
+obj-$(CONFIG_CONTAINER_MEMCTLR) += memctlr.o
diff -puN /dev/null mm/memctlr.c
--- /dev/null 2007-02-02 22:51:23.000000000 +0530
+++ linux-2.6.20-balbir/mm/memctlr.c 2007-02-16 00:22:11.000000000 +0530
@@ -0,0 +1,169 @@
+/*
+ * memctlr.c - Memory Controller for containers
+ *
+ * Copyright (C) Balbir Singh,   IBM Corp. 2006-2007
+ *
+ * This program is free software; you can redistribute it and/or modify it
+ * under the terms of version 2.1 of the GNU Lesser General Public License
+ * as published by the Free Software Foundation.
+ *
+ * This program is distributed in the hope that it would be useful, but
+ * WITHOUT ANY WARRANTY; without even the implied warranty of
+ * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
+ */
+
+#include <linux/init.h>
+#include <linux/parser.h>
+#include <linux/fs.h>
+#include <linux/container.h>
+#include <linux/memctlr.h>
+
+#include <asm/uaccess.h>
+
+#define RES_USAGE_NO_LIMIT   0
+static const char version[] = "0.1";
+
+struct res_counter {
+ unsigned long usage; /* The current usage of the resource being */
+    /* counted        */
+ unsigned long limit; /* The limit on the resource     */
+ unsigned long nr_limit_exceeded;
+};
+
+struct memctlr {
+ struct container_subsys_state css;
+ struct res_counter   counter;
+ spinlock_t    lock;
+};
+
+static struct container_subsys memctlr_subsys;
```

```
+
+static inline struct memctlr *memctlr_from_cont(struct container *cont)
+{
+ return container_of(container_subsys_state(cont, &memctlr_subsys),
+    struct memctlr, css);
+}
+
+static inline struct memctlr *memctlr_from_task(struct task_struct *p)
+{
+ return memctlr_from_cont(task_container(p, &memctlr_subsys));
+}
+
+static int memctlr_create(struct container_subsys *ss, struct container *cont)
+{
+ struct memctlr *mem = kzalloc(sizeof(*mem), GFP_KERNEL);
+ if (!mem)
+  return -ENOMEM;
+
+ spin_lock_init(&mem->lock);
+ cont->subsys[memctlr_subsys.subsys_id] = &mem->css;
+ return 0;
+}
+
+static void memctlr_destroy(struct container_subsys *ss,
+    struct container *cont)
+{
+ kfree(memctlr_from_cont(cont));
+}
+
+static ssize_t memctlr_write(struct container *cont, struct cftype *cft,
+    struct file *file, const char __user *userbuf,
+    size_t nbytes, loff_t *ppos)
+{
+ char *buffer;
+ int ret = 0;
+ unsigned long limit;
+ struct memctlr *mem = memctlr_from_cont(cont);
+
+ BUG_ON(!mem);
+ if ((buffer = kmalloc(nbytes + 1, GFP_KERNEL)) == 0)
+  return -ENOMEM;
+
+ buffer[nbytes] = 0;
+ if (copy_from_user(buffer, userbuf, nbytes)) {
+  ret = -EFAULT;
+  goto out_err;
+ }
+
```

```
+ container_manage_lock();
+ if (container_is_removed(cont)) {
+  ret = -ENODEV;
+  goto out_unlock;
+ }
+
+ limit = simple_strtoul(buffer, NULL, 10);
+ /*
+  * 0 is a valid limit (unlimited resource usage)
+  */
+ if (!limit && strcmp(buffer, "0"))
+  goto out_unlock;
+
+ spin_lock(&mem->lock);
+ mem->counter.limit = limit;
+ spin_unlock(&mem->lock);
+
+ ret = nbytes;
+out_unlock:
+ container_manage_unlock();
+out_err:
+ kfree(buffer);
+ return ret;
+}
+
+static ssize_t memctlr_read(struct container *cont, struct cftype *cft,
+    struct file *file, char __user *userbuf,
+    size_t nbytes, loff_t *ppos)
+{
+ unsigned long usage, limit;
+ char usagebuf[64];  /* Move away from stack later */
+ char *s = usagebuf;
+ struct memctlr *mem = memctlr_from_cont(cont);
+
+ spin_lock(&mem->lock);
+ usage = mem->counter.usage;
+ limit = mem->counter.limit;
+ spin_unlock(&mem->lock);
+
+ s += sprintf(s, "usage %lu, limit %ld\n", usage, limit);
+ return simple_read_from_buffer(userbuf, nbytes, ppos, usagebuf,
+     s - usagebuf);
+}
+
+static struct cftype memctlr_usage = {
+ .name = "memctlr_usage",
+ .read = memctlr_read,
+};
```

```
+
+static struct cftype memctlr_limit = {
+ .name = "memctlr_limit",
+ .write = memctlr_write,
+};
+
+static int memctlr_populate(struct container_subsys *ss,
+    struct container *cont)
+{
+ int rc;
+ if ((rc = container_add_file(cont, &memctlr_usage)) < 0)
+  return rc;
+ if ((rc = container_add_file(cont, &memctlr_limit)) < 0)
+  return rc;
+ return 0;
+}
+
+static struct container_subsys memctlr_subsys = {
+ .name = "memctlr",
+ .create = memctlr_create,
+ .destroy = memctlr_destroy,
+ .populate = memctlr_populate,
+};
+
+int __init memctlr_init(void)
+{
+ int id;
+
+ id = container_register_subsys(&memctlr_subsys);
+ printk("Initializing memctlr version %s, id %d\n", version, id);
+ return id < 0 ? id : 0;
+}
+
+module_init(memctlr_init);
```

–

--
Warm Regards,
Balbir Singh

---

## Subject: [RFC][PATCH][2/4] Add RSS accounting and control
Posted by Balbir Singh on Mon, 19 Feb 2007 06:50:34 GMT
View Forum Message <> Reply to Message

This patch adds the basic accounting hooks to account for pages allocated
into the RSS of a process. Accounting is maintained at two levels, in
the mm_struct of each task and in the memory controller data structure

associated with each node in the container.

When the limit specified for the container is exceeded, the task is killed. RSS accounting is consistent with the current definition of RSS in the kernel. Shared pages are accounted into the RSS of each process as is done in the kernel currently. The code is flexible in that it can be easily modified to work with any definition of RSS.

Signed-off-by: <balbir@in.ibm.com>
---

```
 fs/exec.c              |    4 +
 include/linux/memctlr.h |   38 ++++++++++++
 include/linux/sched.h   |   11 +++
 kernel/fork.c          |   10 +++
 mm/memctlr.c           |  148 +++++++++++++++++++++++++++++++++++++++++++------
 mm/memory.c            |   33 +++++++++-
 mm/rmap.c              |    5 +
 mm/swapfile.c          |    2
 8 files changed, 232 insertions(+), 19 deletions(-)

diff -puN fs/exec.c~memctlr-acct fs/exec.c
--- linux-2.6.20/fs/exec.c~memctlr-acct 2007-02-18 22:55:50.000000000 +0530
+++ linux-2.6.20-balbir/fs/exec.c 2007-02-18 22:55:50.000000000 +0530
@@ -50,6 +50,7 @@
 #include <linux/tsacct_kern.h>
 #include <linux/cn_proc.h>
 #include <linux/audit.h>
+#include <linux/memctlr.h>

 #include <asm/uaccess.h>
 #include <asm/mmu_context.h>
@@ -313,6 +314,9 @@ void install_arg_page(struct vm_area_str
  if (unlikely(anon_vma_prepare(vma)))
   goto out;

+ if (!memctlr_update_rss(mm, 1, MEMCTLR_CHECK_LIMIT))
+  goto out;
+
  flush_dcache_page(page);
  pte = get_locked_pte(mm, address, &ptl);
  if (!pte)
diff -puN include/linux/memctlr.h~memctlr-acct include/linux/memctlr.h
--- linux-2.6.20/include/linux/memctlr.h~memctlr-acct 2007-02-18 22:55:50.000000000 +0530
+++ linux-2.6.20-balbir/include/linux/memctlr.h 2007-02-18 23:28:16.000000000 +0530
@@ -14,9 +14,47 @@
 #ifndef _LINUX_MEMCTLR_H
 #define _LINUX_MEMCTLR_H
```

```
+enum {
+ MEMCTLR_CHECK_LIMIT = true,
+ MEMCTLR_DONT_CHECK_LIMIT = false,
+};
+
 #ifdef CONFIG_CONTAINER_MEMCTLR

+struct res_counter {
+ atomic_long_t usage; /* The current usage of the resource being */
+    /* counted        */
+ atomic_long_t limit; /* The limit on the resource     */
+ atomic_long_t nr_limit_exceeded;
+};
+
+extern int memctlr_mm_init(struct mm_struct *mm);
+extern void memctlr_mm_free(struct mm_struct *mm);
+extern void memctlr_mm_assign_container(struct mm_struct *mm,
+      struct task_struct *p);
+extern int memctlr_update_rss(struct mm_struct *mm, int count, bool check);
+
 #else /* CONFIG_CONTAINER_MEMCTLR  */

+static inline int memctlr_mm_init(struct mm_struct *mm)
+{
+ return 0;
+}
+
+static inline void memctlr_mm_free(struct mm_struct *mm)
+{
+}
+
+static inline void memctlr_mm_assign_container(struct mm_struct *mm,
+      struct task_struct *p)
+{
+}
+
+static inline int memctlr_update_rss(struct mm_struct *mm, int count,
+     bool check)
+{
+ return 0;
+}
+
 #endif /* CONFIG_CONTAINER_MEMCTLR */
 #endif /* _LINUX_MEMCTLR_H */
diff -puN include/linux/sched.h~memctlr-acct include/linux/sched.h
--- linux-2.6.20/include/linux/sched.h~memctlr-acct 2007-02-18 22:55:50.000000000 +0530
+++ linux-2.6.20-balbir/include/linux/sched.h 2007-02-18 22:57:03.000000000 +0530
```

```
@@ -83,6 +83,7 @@ struct sched_param {
 #include <linux/timer.h>
 #include <linux/hrtimer.h>
 #include <linux/task_io_accounting.h>
+#include <linux/memctlr.h>

 #include <asm/processor.h>

@@ -373,6 +374,16 @@ struct mm_struct {
  /* aio bits */
  rwlock_t  ioctx_list_lock;
  struct kioctx  *ioctx_list;
+#ifdef CONFIG_CONTAINER_MEMCTLR
+ /*
+  * Each mm_struct's container, sums up in the container's counter
+  * We can extend this such that, VMA's counters sum up into this
+  * counter
+  */
+ struct res_counter *counter;
+ struct container *container;
+ rwlock_t   container_lock;
+#endif /* CONFIG_CONTAINER_MEMCTLR */
 };

 struct sighand_struct {
diff -puN kernel/fork.c~memctlr-acct kernel/fork.c
--- linux-2.6.20/kernel/fork.c~memctlr-acct 2007-02-18 22:55:50.000000000 +0530
+++ linux-2.6.20-balbir/kernel/fork.c 2007-02-18 22:55:50.000000000 +0530
@@ -50,6 +50,7 @@
 #include <linux/taskstats_kern.h>
 #include <linux/random.h>
 #include <linux/numtasks.h>
+#include <linux/memctlr.h>

 #include <asm/pgtable.h>
 #include <asm/pgalloc.h>
@@ -342,10 +343,15 @@ static struct mm_struct * mm_init(struct
  mm->free_area_cache = TASK_UNMAPPED_BASE;
  mm->cached_hole_size = ~0UL;

+ if (!memctlr_mm_init(mm))
+  goto err;
+
  if (likely(!mm_alloc_pgd(mm))) {
   mm->def_flags = 0;
   return mm;
  }
+
```

```
+err:
  free_mm(mm);
  return NULL;
 }
@@ -361,6 +367,8 @@ struct mm_struct * mm_alloc(void)
  if (mm) {
   memset(mm, 0, sizeof(*mm));
   mm = mm_init(mm);
+  if (mm)
+   memctlr_mm_assign_container(mm, current);
  }
  return mm;
 }
@@ -375,6 +383,7 @@ void fastcall __mmdrop(struct mm_struct
  BUG_ON(mm == &init_mm);
  mm_free_pgd(mm);
  destroy_context(mm);
+ memctlr_mm_free(mm);
  free_mm(mm);
 }

@@ -500,6 +509,7 @@ static struct mm_struct *dup_mm(struct t
  if (init_new_context(tsk, mm))
   goto fail_nocontext;

+ memctlr_mm_assign_container(mm, tsk);
  err = dup_mmap(mm, oldmm);
  if (err)
   goto free_pt;
diff -puN mm/memctlr.c~memctlr-acct mm/memctlr.c
--- linux-2.6.20/mm/memctlr.c~memctlr-acct 2007-02-18 22:55:50.000000000 +0530
+++ linux-2.6.20-balbir/mm/memctlr.c 2007-02-18 23:29:09.000000000 +0530
@@ -23,13 +23,6 @@
 #define RES_USAGE_NO_LIMIT   0
 static const char version[] = "0.1";

-struct res_counter {
- unsigned long usage; /* The current usage of the resource being */
-    /* counted        */
- unsigned long limit; /* The limit on the resource     */
- unsigned long nr_limit_exceeded;
-};
-
 struct memctlr {
  struct container_subsys_state css;
  struct res_counter   counter;
@@ -49,6 +42,74 @@ static inline struct memctlr *memctlr_fr
  return memctlr_from_cont(task_container(p, &memctlr_subsys));
```

```
 }

+int memctlr_mm_init(struct mm_struct *mm)
+{
+ mm->counter = kmalloc(sizeof(struct res_counter), GFP_KERNEL);
+ if (!mm->counter)
+  return 0;
+ atomic_long_set(&mm->counter->usage, 0);
+ atomic_long_set(&mm->counter->limit, 0);
+ rwlock_init(&mm->container_lock);
+ return 1;
+}
+
+void memctlr_mm_free(struct mm_struct *mm)
+{
+ kfree(mm->counter);
+}
+
+static inline void memctlr_mm_assign_container_direct(struct mm_struct *mm,
+       struct container *cont)
+{
+ write_lock(&mm->container_lock);
+ mm->container = cont;
+ write_unlock(&mm->container_lock);
+}
+
+void memctlr_mm_assign_container(struct mm_struct *mm, struct task_struct *p)
+{
+ struct container *cont = task_container(p, &memctlr_subsys);
+ struct memctlr *mem = memctlr_from_cont(cont);
+
+ BUG_ON(!mem);
+ write_lock(&mm->container_lock);
+ mm->container = cont;
+ write_unlock(&mm->container_lock);
+}
+
+/*
+ * Update the rss usage counters for the mm_struct and the container it belongs
+ * to. We do not fail rss for pages shared during fork (see copy_one_pte()).
+ */
+int memctlr_update_rss(struct mm_struct *mm, int count, bool check)
+{
+ int ret = 1;
+ struct container *cont;
+ long usage, limit;
+ struct memctlr *mem;
+
```

```
+ read_lock(&mm->container_lock);
+ cont = mm->container;
+ read_unlock(&mm->container_lock);
+
+ if (!cont)
+  goto done;
+
+ mem = memctlr_from_cont(cont);
+ usage = atomic_long_read(&mem->counter.usage);
+ limit = atomic_long_read(&mem->counter.limit);
+ usage += count;
+ if (check && limit && (usage > limit))
+  ret = 0; /* Above limit, fail */
+ else {
+  atomic_long_add(count, &mem->counter.usage);
+  atomic_long_add(count, &mm->counter->usage);
+ }
+
+done:
+ return ret;
+}
+
 static int memctlr_create(struct container_subsys *ss, struct container *cont)
 {
  struct memctlr *mem = kzalloc(sizeof(*mem), GFP_KERNEL);
@@ -57,6 +118,8 @@ static int memctlr_create(struct contain

  spin_lock_init(&mem->lock);
  cont->subsys[memctlr_subsys.subsys_id] = &mem->css;
+ atomic_long_set(&mem->counter.usage, 0);
+ atomic_long_set(&mem->counter.limit, 0);
  return 0;
 }

@@ -98,9 +161,7 @@ static ssize_t memctlr_write(struct cont
  if (!limit && strcmp(buffer, "0"))
   goto out_unlock;

- spin_lock(&mem->lock);
- mem->counter.limit = limit;
- spin_unlock(&mem->lock);
+ atomic_long_set(&mem->counter.limit, limit);

  ret = nbytes;
 out_unlock:
@@ -114,17 +175,15 @@ static ssize_t memctlr_read(struct conta
     struct file *file, char __user *userbuf,
     size_t nbytes, loff_t *ppos)
```

```
 {
- unsigned long usage, limit;
+ long usage, limit;
  char usagebuf[64];  /* Move away from stack later */
  char *s = usagebuf;
  struct memctlr *mem = memctlr_from_cont(cont);

- spin_lock(&mem->lock);
- usage = mem->counter.usage;
- limit = mem->counter.limit;
- spin_unlock(&mem->lock);
+ usage = atomic_long_read(&mem->counter.usage);
+ limit = atomic_long_read(&mem->counter.limit);

- s += sprintf(s, "usage %lu, limit %ld\n", usage, limit);
+ s += sprintf(s, "usage %ld, limit %ld\n", usage, limit);
  return simple_read_from_buffer(userbuf, nbytes, ppos, usagebuf,
      s - usagebuf);
 }
@@ -150,11 +209,68 @@ static int memctlr_populate(struct conta
  return 0;
 }

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

```
+ * This routine decides how task movement across containers is handled
+ * The simplest strategy is to just move the task (without carrying any old
+ * baggage) The other possibility is move over last accounting information
+ * from mm_struct and charge the new container
+ */
+static void memctlr_attach(struct container_subsys *ss,
+    struct container *cont,
+    struct container *old_cont,
+    struct task_struct *p)
+{
+ struct memctlr *mem, *old_mem;
+ long usage;
+
+ mem = memctlr_from_cont(cont);
+ old_mem = memctlr_from_cont(old_cont);
+
+ memctlr_double_lock(mem, old_mem);
+
+ memctlr_mm_assign_container_direct(p->mm, cont);
+ usage = atomic_read(&p->mm->counter->usage);
+ /*
+  * NOTE: we do not fail the movement in case the addition of a new
+  * task, puts the container overlimit. We reclaim and try our best
+  * to push back the usage of the container.
+  */
+ atomic_long_add(usage, &mem->counter.usage);
+ atomic_long_sub(usage, &old_mem->counter.usage);
+
+ memctlr_double_unlock(mem, old_mem);
+}
+
 static struct container_subsys memctlr_subsys = {
  .name = "memctlr",
  .create = memctlr_create,
  .destroy = memctlr_destroy,
  .populate = memctlr_populate,
+ .attach = memctlr_attach,
 };

 int __init memctlr_init(void)
diff -puN mm/memory.c~memctlr-acct mm/memory.c
--- linux-2.6.20/mm/memory.c~memctlr-acct 2007-02-18 22:55:50.000000000 +0530
+++ linux-2.6.20-balbir/mm/memory.c 2007-02-18 22:55:50.000000000 +0530
@@ -50,6 +50,7 @@
 #include <linux/delayacct.h>
 #include <linux/init.h>
 #include <linux/writeback.h>
+#include <linux/memctlr.h>
```

```
 #include <asm/pgalloc.h>
 #include <asm/uaccess.h>
@@ -532,6 +533,7 @@ again:
  spin_unlock(src_ptl);
  pte_unmap_nested(src_pte - 1);
  add_mm_rss(dst_mm, rss[0], rss[1]);
+ memctlr_update_rss(dst_mm, rss[0] + rss[1], MEMCTLR_DONT_CHECK_LIMIT);
  pte_unmap_unlock(dst_pte - 1, dst_ptl);
  cond_resched();
  if (addr != end)
@@ -1128,6 +1130,7 @@ static int zeromap_pte_range(struct mm_s
   page_cache_get(page);
   page_add_file_rmap(page);
   inc_mm_counter(mm, file_rss);
+  memctlr_update_rss(mm, 1, MEMCTLR_DONT_CHECK_LIMIT);
   set_pte_at(mm, addr, pte, zero_pte);
  } while (pte++, addr += PAGE_SIZE, addr != end);
  arch_leave_lazy_mmu_mode();
@@ -1223,6 +1226,10 @@ static int insert_page(struct mm_struct
  if (PageAnon(page))
   goto out;
  retval = -ENOMEM;
+
+ if (!memctlr_update_rss(mm, 1, MEMCTLR_CHECK_LIMIT))
+  goto out;
+
  flush_dcache_page(page);
  pte = get_locked_pte(mm, addr, &ptl);
  if (!pte)
@@ -1580,6 +1587,9 @@ gotten:
   cow_user_page(new_page, old_page, address, vma);
  }

+ if (!memctlr_update_rss(mm, 1, MEMCTLR_CHECK_LIMIT))
+  goto oom;
+
  /*
   * Re-check the pte - we dropped the lock
   */
@@ -1612,7 +1622,9 @@ gotten:
   /* Free the old page.. */
   new_page = old_page;
   ret |= VM_FAULT_WRITE;
- }
+ } else
+  memctlr_update_rss(mm, -1, MEMCTLR_DONT_CHECK_LIMIT);
+
```

```
  if (new_page)
   page_cache_release(new_page);
  if (old_page)
@@ -2024,16 +2036,19 @@ static int do_swap_page(struct mm_struct
  mark_page_accessed(page);
  lock_page(page);

+ if (!memctlr_update_rss(mm, 1, MEMCTLR_CHECK_LIMIT))
+  goto out_nomap;
+
  /*
   * Back out if somebody else already faulted in this pte.
   */
  page_table = pte_offset_map_lock(mm, pmd, address, &ptl);
  if (unlikely(!pte_same(*page_table, orig_pte)))
-  goto out_nomap;
+  goto out_nomap_uncharge;

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

  /* The page isn't present yet, go ahead with the fault. */
@@ -2068,6 +2083,8 @@ unlock:
  pte_unmap_unlock(page_table, ptl);
 out:
  return ret;
+out_nomap_uncharge:
+ memctlr_update_rss(mm, -1, MEMCTLR_DONT_CHECK_LIMIT);
 out_nomap:
  pte_unmap_unlock(page_table, ptl);
  unlock_page(page);
@@ -2092,6 +2109,9 @@ static int do_anonymous_page(struct mm_s
   /* Allocate our own private page. */
   pte_unmap(page_table);

+  if (!memctlr_update_rss(mm, 1, MEMCTLR_CHECK_LIMIT))
+   goto oom;
+
   if (unlikely(anon_vma_prepare(vma)))
    goto oom;
   page = alloc_zeroed_user_highpage(vma, address);
@@ -2108,6 +2128,8 @@ static int do_anonymous_page(struct mm_s
   lru_cache_add_active(page);
   page_add_new_anon_rmap(page, vma, address);
  } else {
```

```
+  memctlr_update_rss(mm, 1, MEMCTLR_DONT_CHECK_LIMIT);
+
   /* Map the ZERO_PAGE - vm_page_prot is readonly */
   page = ZERO_PAGE(address);
   page_cache_get(page);
@@ -2218,6 +2240,9 @@ retry:
   }
  }

+ if (!memctlr_update_rss(mm, 1, MEMCTLR_CHECK_LIMIT))
+  goto oom;
+
  page_table = pte_offset_map_lock(mm, pmd, address, &ptl);
  /*
   * For a file-backed vma, someone could have truncated or otherwise
@@ -2227,6 +2252,7 @@ retry:
  if (mapping && unlikely(sequence != mapping->truncate_count)) {
   pte_unmap_unlock(page_table, ptl);
   page_cache_release(new_page);
+  memctlr_update_rss(mm, -1, MEMCTLR_DONT_CHECK_LIMIT);
   cond_resched();
   sequence = mapping->truncate_count;
   smp_rmb();
@@ -2265,6 +2291,7 @@ retry:
  } else {
   /* One of our sibling threads was faster, back out. */
   page_cache_release(new_page);
+  memctlr_update_rss(mm, -1, MEMCTLR_DONT_CHECK_LIMIT);
   goto unlock;
  }

diff -puN mm/rmap.c~memctlr-acct mm/rmap.c
--- linux-2.6.20/mm/rmap.c~memctlr-acct 2007-02-18 22:55:50.000000000 +0530
+++ linux-2.6.20-balbir/mm/rmap.c 2007-02-18 23:28:16.000000000 +0530
@@ -602,6 +602,11 @@ void page_remove_rmap(struct page *page,
   __dec_zone_page_state(page,
     PageAnon(page) ? NR_ANON_PAGES : NR_FILE_MAPPED);
  }
+ /*
+  * When we pass MEMCTLR_DONT_CHECK_LIMIT, it is ok to call
+  * this function under the pte lock (since we will not block in reclaim)
+  */
+ memctlr_update_rss(vma->vm_mm, -1, MEMCTLR_DONT_CHECK_LIMIT);
 }

 /*
diff -puN mm/swapfile.c~memctlr-acct mm/swapfile.c
--- linux-2.6.20/mm/swapfile.c~memctlr-acct 2007-02-18 22:55:50.000000000 +0530
```

```
+++ linux-2.6.20-balbir/mm/swapfile.c 2007-02-18 22:55:50.000000000 +0530
@@ -27,6 +27,7 @@
 #include <linux/mutex.h>
 #include <linux/capability.h>
 #include <linux/syscalls.h>
+#include <linux/memctlr.h>

 #include <asm/pgtable.h>
 #include <asm/tlbflush.h>
@@ -514,6 +515,7 @@ static void unuse_pte(struct vm_area_str
  set_pte_at(vma->vm_mm, addr, pte,
     pte_mkold(mk_pte(page, vma->vm_page_prot)));
  page_add_anon_rmap(page, vma, addr);
+ memctlr_update_rss(vma->vm_mm, 1, MEMCTLR_DONT_CHECK_LIMIT);
  swap_free(entry);
  /*
   * Move the page to the active list so it is not
_

--
 Warm Regards,
 Balbir Singh
```

---

Subject: [RFC][PATCH][3/4] Add reclaim support
Posted by Balbir Singh on Mon, 19 Feb 2007 06:50:42 GMT
View Forum Message <> Reply to Message

This patch reclaims pages from a container when the container limit is hit.
The executable is oom'ed only when the container it is running in, is overlimit
and we could not reclaim any pages belonging to the container

A parameter called pushback, controls how much memory is reclaimed when the
limit is hit. It should be easy to expose this knob to user space, but
currently it is hard coded to 20% of the total limit of the container.

isolate_lru_pages() has been modified to isolate pages belonging to a
particular container, so that reclaim code will reclaim only container
pages. For shared pages, reclaim does not unmap all mappings of the page,
it only unmaps those mappings that are over their limit. This ensures
that other containers are not penalized while reclaiming shared pages.

Parallel reclaim per container is not allowed. Each controller has a wait
queue that ensures that only one task per control is running reclaim on
that container.

Signed-off-by: <balbir@in.ibm.com>

```
---

 include/linux/memctlr.h |    8 ++
 include/linux/rmap.h    |   13 +++-
 include/linux/swap.h    |    4 +
 mm/memctlr.c            |  137 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++----
 mm/migrate.c            |    2
 mm/rmap.c               |   96 ++++++++++++++++++++++++++++++++++++++++++++++++++--
 mm/vmscan.c             |   94 ++++++++++++++++++++++++++++++++++++++++++++++++++++----
 7 files changed, 324 insertions(+), 30 deletions(-)

diff -puN include/linux/memctlr.h~memctlr-reclaim-on-limit include/linux/memctlr.h
---  linux-2.6.20/include/linux/memctlr.h~memctlr-reclaim-on-limi t 2007-02-18
23:29:14.000000000 +0530
+++ linux-2.6.20-balbir/include/linux/memctlr.h 2007-02-18 23:29:14.000000000 +0530
@@ -20,6 +20,7 @@ enum {
 };

 #ifdef CONFIG_CONTAINER_MEMCTLR
+#include <linux/wait.h>

 struct res_counter {
  atomic_long_t usage; /* The current usage of the resource being */
@@ -33,6 +34,9 @@ extern void memctlr_mm_free(struct mm_st
 extern void memctlr_mm_assign_container(struct mm_struct *mm,
       struct task_struct *p);
 extern int memctlr_update_rss(struct mm_struct *mm, int count, bool check);
+extern int memctlr_mm_overlimit(struct mm_struct *mm, void *sc_cont);
+extern wait_queue_head_t memctlr_reclaim_wq;
+extern bool memctlr_reclaim_in_progress;

 #else /* CONFIG_CONTAINER_MEMCTLR  */

@@ -56,5 +60,9 @@ static inline int memctlr_update_rss(str
  return 0;
 }

+int memctlr_mm_overlimit(struct mm_struct *mm, void *sc_cont)
+{
+ return 0;
+}
 #endif /* CONFIG_CONTAINER_MEMCTLR */
 #endif /* _LINUX_MEMCTLR_H */
diff -puN include/linux/rmap.h~memctlr-reclaim-on-limit include/linux/rmap.h
--- linux-2.6.20/include/linux/rmap.h~memctlr-reclaim-on-limit 2007-02-18 23:29:14.000000000
+0530
+++ linux-2.6.20-balbir/include/linux/rmap.h 2007-02-18 23:29:14.000000000 +0530
@@ -90,7 +90,15 @@ static inline void page_dup_rmap(struct
```

```
  * Called from mm/vmscan.c to handle paging out
  */
 int page_referenced(struct page *, int is_locked);
-int try_to_unmap(struct page *, int ignore_refs);
+int try_to_unmap(struct page *, int ignore_refs, void *container);
+#ifdef CONFIG_CONTAINER_MEMCTLR
+bool page_in_container(struct page *page, struct zone *zone, void *container);
+#else
+static inline bool page_in_container(struct page *page, struct zone *zone, void *container)
+{
+ return true;
+}
+#endif /* CONFIG_CONTAINER_MEMCTLR */

 /*
  * Called from mm/filemap_xip.c to unmap empty zero page
@@ -118,7 +126,8 @@ int page_mkclean(struct page *);
 #define anon_vma_link(vma) do {} while (0)

 #define page_referenced(page,l) TestClearPageReferenced(page)
-#define try_to_unmap(page, refs) SWAP_FAIL
+#define try_to_unmap(page, refs, container) SWAP_FAIL
+#define page_in_container(page, zone, container)  true

 static inline int page_mkclean(struct page *page)
 {
diff -puN include/linux/swap.h~memctlr-reclaim-on-limit include/linux/swap.h
--- linux-2.6.20/include/linux/swap.h~memctlr-reclaim-on-limit 2007-02-18 23:29:14.000000000
+0530
+++ linux-2.6.20-balbir/include/linux/swap.h 2007-02-18 23:29:14.000000000 +0530
@@ -188,6 +188,10 @@ extern void swap_setup(void);
 /* linux/mm/vmscan.c */
 extern unsigned long try_to_free_pages(struct zone **, gfp_t);
 extern unsigned long shrink_all_memory(unsigned long nr_pages);
+#ifdef CONFIG_CONTAINER_MEMCTLR
+extern unsigned long memctlr_shrink_mapped_memory(unsigned long nr_pages,
+       void *container);
+#endif
 extern int vm_swappiness;
 extern int remove_mapping(struct address_space *mapping, struct page *page);
 extern long vm_total_pages;
diff -puN mm/memctlr.c~memctlr-reclaim-on-limit mm/memctlr.c
--- linux-2.6.20/mm/memctlr.c~memctlr-reclaim-on-limit 2007-02-18 23:29:14.000000000 +0530
+++ linux-2.6.20-balbir/mm/memctlr.c 2007-02-18 23:34:51.000000000 +0530
@@ -17,16 +17,26 @@
 #include <linux/fs.h>
 #include <linux/container.h>
 #include <linux/memctlr.h>
```

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

 #include <asm/uaccess.h>

-#define RES_USAGE_NO_LIMIT   0
+#define RES_USAGE_NO_LIMIT 0
 static const char version[] = "0.1";
+/*
+ * Explore exporting these knobs to user space
+ */
+static const int pushback = 20;  /* What percentage of memory to reclaim */
+static const int nr_retries = 5; /* How many times do we try to reclaim  */
+
+static atomic_t nr_reclaim;

 struct memctlr {
  struct container_subsys_state css;
  struct res_counter   counter;
  spinlock_t    lock;
+ wait_queue_head_t  wq;
+ bool    reclaim_in_progress;
 };

 static struct container_subsys memctlr_subsys;
@@ -42,6 +52,44 @@ static inline struct memctlr *memctlr_fr
  return memctlr_from_cont(task_container(p, &memctlr_subsys));
 }

+/*
+ * checks if the mm's container and scan control passed container match, if
+ * so, is the container over it's limit. Returns 1 if the container is above
+ * its limit.
+ */
+int memctlr_mm_overlimit(struct mm_struct *mm, void *sc_cont)
+{
+ struct container *cont;
+ struct memctlr *mem;
+ long usage, limit;
+ int ret = 1;
+
+ if (!sc_cont)
+  goto out;
+
+ read_lock(&mm->container_lock);
+ cont = mm->container;
+
+ /*
+  * Regular reclaim, let it proceed as usual
```

```
+   */
+ if (!sc_cont)
+  goto out;
+
+ ret = 0;
+ if (cont != sc_cont)
+  goto out;
+
+ mem = memctlr_from_cont(cont);
+ usage = atomic_long_read(&mem->counter.usage);
+ limit = atomic_long_read(&mem->counter.limit);
+ if (limit && (usage > limit))
+  ret = 1;
+out:
+ read_unlock(&mm->container_lock);
+ return ret;
+}
+
 int memctlr_mm_init(struct mm_struct *mm)
 {
  mm->counter = kmalloc(sizeof(struct res_counter), GFP_KERNEL);
@@ -77,6 +125,46 @@ void memctlr_mm_assign_container(struct
  write_unlock(&mm->container_lock);
 }

+static int memctlr_check_and_reclaim(struct container *cont, long usage,
+     long limit)
+{
+ unsigned long nr_pages = 0;
+ unsigned long nr_reclaimed = 0;
+ int retries = nr_retries;
+ int ret = 1;
+ struct memctlr *mem;
+
+ mem = memctlr_from_cont(cont);
+ spin_lock(&mem->lock);
+ while ((retries-- > 0) && limit && (usage > limit)) {
+  if (mem->reclaim_in_progress) {
+   spin_unlock(&mem->lock);
+   wait_event(mem->wq, !mem->reclaim_in_progress);
+   spin_lock(&mem->lock);
+  } else {
+   if (!nr_pages)
+    nr_pages = (pushback * limit) / 100;
+   mem->reclaim_in_progress = true;
+   spin_unlock(&mem->lock);
+   nr_reclaimed += memctlr_shrink_mapped_memory(nr_pages,
+         cont);
```

```
+   spin_lock(&mem->lock);
+   mem->reclaim_in_progress = false;
+   wake_up_all(&mem->wq);
+  }
+  /*
+    * Resample usage and limit after reclaim
+    */
+  usage = atomic_long_read(&mem->counter.usage);
+  limit = atomic_long_read(&mem->counter.limit);
+ }
+ spin_unlock(&mem->lock);
+
+ if (limit && (usage > limit))
+  ret = 0;
+ return ret;
+}
+
 /*
  * Update the rss usage counters for the mm_struct and the container it belongs
  * to. We do not fail rss for pages shared during fork (see copy_one_pte()).
@@ -99,13 +187,14 @@ int memctlr_update_rss(struct mm_struct
  usage = atomic_long_read(&mem->counter.usage);
  limit = atomic_long_read(&mem->counter.limit);
  usage += count;
- if (check && limit && (usage > limit))
-  ret = 0; /* Above limit, fail */
- else {
-  atomic_long_add(count, &mem->counter.usage);
-  atomic_long_add(count, &mm->counter->usage);
- }

+ if (check) {
+  ret = memctlr_check_and_reclaim(cont, usage, limit);
+  if (!ret)
+   goto done;
+ }
+ atomic_long_add(count, &mem->counter.usage);
+ atomic_long_add(count, &mm->counter->usage);
 done:
  return ret;
 }
@@ -120,6 +209,8 @@ static int memctlr_create(struct contain
  cont->subsys[memctlr_subsys.subsys_id] = &mem->css;
  atomic_long_set(&mem->counter.usage, 0);
  atomic_long_set(&mem->counter.limit, 0);
+ init_waitqueue_head(&mem->wq);
+ mem->reclaim_in_progress = 0;
  return 0;
```

```
 }

@@ -134,8 +225,8 @@ static ssize_t memctlr_write(struct cont
     size_t nbytes, loff_t *ppos)
 {
  char *buffer;
- int ret = 0;
- unsigned long limit;
+ int ret = nbytes;
+ unsigned long cur_limit, limit, usage;
  struct memctlr *mem = memctlr_from_cont(cont);

  BUG_ON(!mem);
@@ -162,8 +253,16 @@ static ssize_t memctlr_write(struct cont
   goto out_unlock;

  atomic_long_set(&mem->counter.limit, limit);
+ usage = atomic_read(&mem->counter.usage);
+ cur_limit = atomic_read(&mem->counter.limit);
+ if (limit && (usage > limit)) {
+  ret = memctlr_check_and_reclaim(cont, usage, cur_limit);
+  if (!ret) {
+   ret = -EAGAIN; /* Try again, later */
+   goto out_unlock;
+  }
+ }

- ret = nbytes;
 out_unlock:
  container_manage_unlock();
 out_err:
@@ -233,6 +332,17 @@ static inline void memctlr_double_unlock
  }
 }

+int memctlr_can_attach(struct container_subsys *ss, struct container *cont,
+   struct task_struct *p)
+{
+ /*
+  * Allow only the thread group leader to change containers
+  */
+ if (p->pid != p->tgid)
+  return -EINVAL;
+ return 0;
+}
+
 /*
  * This routine decides how task movement across containers is handled
```

```
  * The simplest strategy is to just move the task (without carrying any old
@@ -247,6 +357,12 @@ static void memctlr_attach(struct contai
  struct memctlr *mem, *old_mem;
  long usage;

+ /*
+   * See if this can be stopped at the upper layer
+   */
+ if (cont == old_cont)
+  return;
+
  mem = memctlr_from_cont(cont);
  old_mem = memctlr_from_cont(old_cont);

@@ -278,6 +394,7 @@ int __init memctlr_init(void)
  int id;

  id = container_register_subsys(&memctlr_subsys);
+ atomic_set(&nr_reclaim, 0);
  printk("Initializing memctlr version %s, id %d\n", version, id);
  return id < 0 ? id : 0;
 }
diff -puN mm/migrate.c~memctlr-reclaim-on-limit mm/migrate.c
--- linux-2.6.20/mm/migrate.c~memctlr-reclaim-on-limit 2007-02-18 23:29:14.000000000 +0530
+++ linux-2.6.20-balbir/mm/migrate.c 2007-02-18 23:29:14.000000000 +0530
@@ -623,7 +623,7 @@ static int unmap_and_move(new_page_t get
  /*
   * Establish migration ptes or remove ptes
   */
- try_to_unmap(page, 1);
+ try_to_unmap(page, 1, NULL);
  if (!page_mapped(page))
   rc = move_to_new_page(newpage, page);

diff -puN mm/rmap.c~memctlr-reclaim-on-limit mm/rmap.c
--- linux-2.6.20/mm/rmap.c~memctlr-reclaim-on-limit 2007-02-18 23:29:14.000000000 +0530
+++ linux-2.6.20-balbir/mm/rmap.c 2007-02-18 23:29:14.000000000 +0530
@@ -792,7 +792,7 @@ static void try_to_unmap_cluster(unsigne
  pte_unmap_unlock(pte - 1, ptl);
 }

-static int try_to_unmap_anon(struct page *page, int migration)
+static int try_to_unmap_anon(struct page *page, int migration, void *container)
 {
  struct anon_vma *anon_vma;
  struct vm_area_struct *vma;
@@ -803,6 +803,13 @@ static int try_to_unmap_anon(struct page
   return ret;
```

```
  list_for_each_entry(vma, &anon_vma->head, anon_vma_node) {
+  /*
+    * When reclaiming memory on behalf of overlimit containers
+    * shared pages are spared, they are only unmapped from
+    * the vma's (mm's) whose containers are over limit
+    */
+  if (!memctlr_mm_overlimit(vma->vm_mm, container))
+   continue;
   ret = try_to_unmap_one(page, vma, migration);
   if (ret == SWAP_FAIL || !page_mapped(page))
    break;
@@ -820,7 +827,7 @@ static int try_to_unmap_anon(struct page
  *
  * This function is only called from try_to_unmap for object-based pages.
  */
-static int try_to_unmap_file(struct page *page, int migration)
+static int try_to_unmap_file(struct page *page, int migration, void *container)
 {
  struct address_space *mapping = page->mapping;
  pgoff_t pgoff = page->index << (PAGE_CACHE_SHIFT - PAGE_SHIFT);
@@ -834,6 +841,12 @@ static int try_to_unmap_file(struct page

  spin_lock(&mapping->i_mmap_lock);
  vma_prio_tree_foreach(vma, &iter, &mapping->i_mmap, pgoff, pgoff) {
+  /*
+    * If we are reclaiming memory due to containers being overlimit
+    * and this mm is not over it's limit, spare the page
+    */
+  if (!memctlr_mm_overlimit(vma->vm_mm, container))
+   continue;
   ret = try_to_unmap_one(page, vma, migration);
   if (ret == SWAP_FAIL || !page_mapped(page))
    goto out;
@@ -880,6 +893,8 @@ static int try_to_unmap_file(struct page
       shared.vm_set.list) {
    if ((vma->vm_flags & VM_LOCKED) && !migration)
     continue;
+   if (!memctlr_mm_overlimit(vma->vm_mm, container))
+    continue;
    cursor = (unsigned long) vma->vm_private_data;
    while ( cursor < max_nl_cursor &&
     cursor < vma->vm_end - vma->vm_start) {
@@ -919,19 +934,90 @@ out:
  * SWAP_AGAIN - we missed a mapping, try again later
  * SWAP_FAIL - the page is unswappable
  */
-int try_to_unmap(struct page *page, int migration)
```

```
+int try_to_unmap(struct page *page, int migration, void *container)
 {
  int ret;

  BUG_ON(!PageLocked(page));

  if (PageAnon(page))
-  ret = try_to_unmap_anon(page, migration);
+  ret = try_to_unmap_anon(page, migration, container);
  else
-  ret = try_to_unmap_file(page, migration);
+  ret = try_to_unmap_file(page, migration, container);

  if (!page_mapped(page))
   ret = SWAP_SUCCESS;
  return ret;
 }

+#ifdef CONFIG_CONTAINER_MEMCTLR
+bool anon_page_in_container(struct page *page, void *container)
+{
+ struct anon_vma *anon_vma;
+ struct vm_area_struct *vma;
+ bool ret = false;
+
+ anon_vma = page_lock_anon_vma(page);
+ if (!anon_vma)
+  return ret;
+
+ list_for_each_entry(vma, &anon_vma->head, anon_vma_node)
+  if (memctlr_mm_overlimit(vma->vm_mm, container)) {
+   ret = true;
+   break;
+  }
+
+ spin_unlock(&anon_vma->lock);
+ return ret;
+}
+
+bool file_page_in_container(struct page *page, void *container)
+{
+ bool ret = false;
+ struct vm_area_struct *vma;
+ struct address_space *mapping = page_mapping(page);
+ struct prio_tree_iter iter;
+ pgoff_t pgoff = page->index << (PAGE_CACHE_SHIFT - PAGE_SHIFT);
+
+ if (!mapping)
```

```
+  return ret;
+
+ spin_lock(&mapping->i_mmap_lock);
+
+ vma_prio_tree_foreach(vma, &iter, &mapping->i_mmap, pgoff, pgoff)
+  /*
+    * Check if the page belongs to the container and it is overlimit
+    */
+  if (memctlr_mm_overlimit(vma->vm_mm, container)) {
+   ret = true;
+   goto done;
+  }
+
+ if (list_empty(&mapping->i_mmap_nonlinear))
+  goto done;
+
+ list_for_each_entry(vma, &mapping->i_mmap_nonlinear,
+      shared.vm_set.list)
+  if (memctlr_mm_overlimit(vma->vm_mm, container)) {
+   ret = true;
+   goto done;
+  }
+done:
+ spin_unlock(&mapping->i_mmap_lock);
+ return ret;
+}
+
+bool page_in_container(struct page *page, struct zone *zone, void *container)
+{
+ bool ret;
+
+ spin_unlock_irq(&zone->lru_lock);
+ if (PageAnon(page))
+  ret = anon_page_in_container(page, container);
+ else
+  ret = file_page_in_container(page, container);
+
+ spin_lock_irq(&zone->lru_lock);
+ return ret;
+}
+#endif
diff -puN mm/vmscan.c~memctlr-reclaim-on-limit mm/vmscan.c
--- linux-2.6.20/mm/vmscan.c~memctlr-reclaim-on-limit 2007-02-18 23:29:14.000000000 +0530
+++ linux-2.6.20-balbir/mm/vmscan.c 2007-02-18 23:29:14.000000000 +0530
@@ -42,6 +42,7 @@
 #include <asm/div64.h>

 #include <linux/swapops.h>
```

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

 #include "internal.h"

@@ -66,6 +67,9 @@ struct scan_control {
  int swappiness;

  int all_unreclaimable;
+
+ void *container;  /* Used by containers for reclaiming */
+     /* pages when the limit is exceeded  */
 };

 /*
@@ -507,7 +511,7 @@ static unsigned long shrink_page_list(st
    * processes. Try to unmap it here.
    */
   if (page_mapped(page) && mapping) {
-   switch (try_to_unmap(page, 0)) {
+   switch (try_to_unmap(page, 0, sc->container)) {
    case SWAP_FAIL:
     goto activate_locked;
    case SWAP_AGAIN:
@@ -621,13 +625,15 @@ keep:
  */
 static unsigned long isolate_lru_pages(unsigned long nr_to_scan,
   struct list_head *src, struct list_head *dst,
-  unsigned long *scanned)
+  unsigned long *scanned, struct zone *zone, void *container,
+  unsigned long max_scan)
 {
  unsigned long nr_taken = 0;
  struct page *page;
- unsigned long scan;
+ unsigned long scan, vscan;

- for (scan = 0; scan < nr_to_scan && !list_empty(src); scan++) {
+ for (scan = 0, vscan = 0; scan < nr_to_scan && (vscan < max_scan) &&
+     !list_empty(src); scan++, vscan++) {
   struct list_head *target;
   page = lru_to_page(src);
   prefetchw_prev_lru_page(page, src, flags);
@@ -636,6 +642,15 @@ static unsigned long isolate_lru_pages(u

   list_del(&page->lru);
   target = src;
+  /*
+    * For containers, do not scan the page unless it
```

```
+    * belongs to the container we are reclaiming for
+    */
+  if (container && !page_in_container(page, zone, container)) {
+   scan--;
+   goto done;
+  }
+
   if (likely(get_page_unless_zero(page))) {
    /*
     * Be careful not to clear PageLRU until after we're
@@ -646,7 +661,7 @@ static unsigned long isolate_lru_pages(u
    target = dst;
    nr_taken++;
   } /* else it is being freed elsewhere */
-
+done:
   list_add(&page->lru, target);
  }

@@ -678,7 +693,8 @@ static unsigned long shrink_inactive_lis

   nr_taken = isolate_lru_pages(sc->swap_cluster_max,
          &zone->inactive_list,
-         &page_list, &nr_scan);
+         &page_list, &nr_scan, zone,
+         sc->container, zone->nr_inactive);
   zone->nr_inactive -= nr_taken;
   zone->pages_scanned += nr_scan;
   spin_unlock_irq(&zone->lru_lock);
@@ -823,7 +839,8 @@ force_reclaim_mapped:
  lru_add_drain();
  spin_lock_irq(&zone->lru_lock);
  pgmoved = isolate_lru_pages(nr_pages, &zone->active_list,
-        &l_hold, &pgscanned);
+        &l_hold, &pgscanned, zone, sc->container,
+        zone->nr_active);
  zone->pages_scanned += pgscanned;
  zone->nr_active -= pgmoved;
  spin_unlock_irq(&zone->lru_lock);
@@ -1361,7 +1378,7 @@ void wakeup_kswapd(struct zone *zone, in
  wake_up_interruptible(&pgdat->kswapd_wait);
 }

-#ifdef CONFIG_PM
+#if defined(CONFIG_PM) || defined(CONFIG_CONTAINER_MEMCTLR)
 /*
  * Helper function for shrink_all_memory().  Tries to reclaim 'nr_pages' pages
  * from LRU lists system-wide, for given pass and priority, and returns the
```

```
@@ -1370,7 +1387,7 @@ void wakeup_kswapd(struct zone *zone, in
  * For pass > 3 we also try to shrink the LRU lists that contain a few pages
  */
 static unsigned long shrink_all_zones(unsigned long nr_pages, int prio,
-          int pass, struct scan_control *sc)
+          int pass, int max_pass, struct scan_control *sc)
 {
  struct zone *zone;
  unsigned long nr_to_scan, ret = 0;
@@ -1386,7 +1403,7 @@ static unsigned long shrink_all_zones(un
   /* For pass = 0 we don't shrink the active list */
   if (pass > 0) {
    zone->nr_scan_active += (zone->nr_active >> prio) + 1;
-   if (zone->nr_scan_active >= nr_pages || pass > 3) {
+   if (zone->nr_scan_active >= nr_pages || pass > max_pass) {
     zone->nr_scan_active = 0;
     nr_to_scan = min(nr_pages, zone->nr_active);
     shrink_active_list(nr_to_scan, zone, sc, prio);
@@ -1394,7 +1411,7 @@ static unsigned long shrink_all_zones(un
   }

   zone->nr_scan_inactive += (zone->nr_inactive >> prio) + 1;
-  if (zone->nr_scan_inactive >= nr_pages || pass > 3) {
+  if (zone->nr_scan_inactive >= nr_pages || pass > max_pass) {
    zone->nr_scan_inactive = 0;
    nr_to_scan = min(nr_pages, zone->nr_inactive);
    ret += shrink_inactive_list(nr_to_scan, zone, sc);
@@ -1405,7 +1422,9 @@ static unsigned long shrink_all_zones(un

  return ret;
 }
+#endif

+#ifdef CONFIG_PM
 static unsigned long count_lru_pages(void)
 {
  struct zone *zone;
@@ -1477,7 +1496,7 @@ unsigned long shrink_all_memory(unsigned
    unsigned long nr_to_scan = nr_pages - ret;

    sc.nr_scanned = 0;
-   ret += shrink_all_zones(nr_to_scan, prio, pass, &sc);
+   ret += shrink_all_zones(nr_to_scan, prio, pass, 3, &sc);
    if (ret >= nr_pages)
     goto out;

@@ -1512,6 +1531,57 @@ out:
 }
```

```
 #endif

+#ifdef CONFIG_CONTAINER_MEMCTLR
+/*
+ * Try to free `nr_pages' of memory, system-wide, and return the number of
+ * freed pages.
+ * Modelled after shrink_all_memory()
+ */
+unsigned long memctlr_shrink_mapped_memory(unsigned long nr_pages, void *container)
+{
+ unsigned long ret = 0;
+ int pass;
+ unsigned long nr_total_scanned = 0;
+
+ struct scan_control sc = {
+  .gfp_mask = GFP_KERNEL,
+  .may_swap = 0,
+  .swap_cluster_max = nr_pages,
+  .may_writepage = 1,
+  .swappiness = vm_swappiness,
+  .container = container,
+  .may_swap = 1,
+  .swappiness = 100,
+ };
+
+ /*
+  * We try to shrink LRUs in 3 passes:
+  * 0 = Reclaim from inactive_list only
+  * 1 = Reclaim mapped (normal reclaim)
+  * 2 = 2nd pass of type 1
+  */
+ for (pass = 0; pass < 3; pass++) {
+  int prio;
+
+  for (prio = DEF_PRIORITY; prio >= 0; prio--) {
+   unsigned long nr_to_scan = nr_pages - ret;
+
+   sc.nr_scanned = 0;
+   ret += shrink_all_zones(nr_to_scan, prio,
+      pass, 1, &sc);
+   if (ret >= nr_pages)
+    goto out;
+
+   nr_total_scanned += sc.nr_scanned;
+   if (sc.nr_scanned && prio < DEF_PRIORITY - 2)
+    congestion_wait(WRITE, HZ / 10);
+  }
+ }
```

```
+out:
+ return ret;
+}
+#endif
+
 /* It's optimal to keep kswapds on the same CPUs as their memory, but
    not required for correctness.  So if the last cpu in a node goes
    away, we get changed to run anywhere: as the first one comes back,
```

–

--
Warm Regards,
Balbir Singh

---

Subject: [RFC][PATCH][4/4] RSS controller documentation
Posted by Balbir Singh on Mon, 19 Feb 2007 06:50:50 GMT
View Forum Message <> Reply to Message

Signed-off-by: <balbir@in.ibm.com>
---

 Documentation/memctlr.txt |   70 ++++++++++++++++++++++++++++++++++++++++++++++
 1 file changed, 70 insertions(+)

```
diff -puN /dev/null Documentation/memctlr.txt
--- /dev/null 2007-02-02 22:51:23.000000000 +0530
+++ linux-2.6.20-balbir/Documentation/memctlr.txt 2007-02-19 00:51:44.000000000 +0530
@@ -0,0 +1,70 @@
+Introduction
+------------
+
+The memory controller is a controller module written under the containers
+framework. It can be used to limit the resource usage of a group of
+tasks grouped by the container.
+
+Accounting
+----------
+
+The memory controller tracks the RSS usage of the tasks in the container.
+The definition of RSS was debated on lkml in the following thread
+
+ http://lkml.org/lkml/2006/10/10/130
+
+This patch is flexible, it is easy to adapt the patch to any definition
+of RSS. The current accounting is based on the current definition of
+RSS. Each page mapped is charged to the container.
+
```

+The accounting is done at two levels, each process has RSS accounting in
+the mm_struct and in the container it belongs to. The mm_struct accounting
+is used when a task switches (migrates to a different) container(s). The
+accounting information for the task is subtracted from the source container
+and added to the destination container. If as result of the migration, the
+destination container goes over limit, no action is taken until some task
+in the destination container runs and tries to map a new page in its
+page table.
+
+The current RSS usage can be seen in the memctlr_usage file. The value
+is in units of pages.
+
+Control
+-------
+
+The memctlr_limit file allows the user to set a limit on the number of
+pages that can be mapped by the processes in the container. A special
+value of 0 (which is the default limit of any new container), indicates
+that the container can use unlimited amount of RSS.
+
+Reclaim
+-------
+
+When the limit set in the container is hit, the memory controller starts
+reclaiming pages belonging to the container (simulating a local LRU in
+some sense). isolate_lru_pages() has been modified to isolate lru
+pages belonging to a specific container. Parallel reclaims on the same
+container are not allowed, other tasks end up waiting for the any existing
+reclaim to finish.
+
+The reclaim code uses two internal knobs, retries and pushback. pushback
+specifies the percentage of memory to be reclaimed when the container goes
+over limit. The retries knob, controls how many times reclaim is retried
+before the task is killed (because reclaim failed).
+
+Shared pages are treated specially during reclaim. They are not force
+reclaimed, they are only unmapped from containers which are over limit.
+This ensures that other containers do not pay a penalty for a shared
+page being reclaimed when a paritcular container goes over its limit.
+
+NOTE: All limits are hard limits.
+
+Future Plans
+------------
+
+The current controller implements only RSS control. It is planned to add
+the following components
+

+1. Page Cache control
+2. mlock'ed memory control
+3. kernel memory allocation control (memory allocated on behalf of a task)

_

--
Warm Regards,
Balbir Singh

---

Subject: Re: [RFC][PATCH][0/4] Memory controller (RSS Control)
Posted by Andrew Morton on Mon, 19 Feb 2007 08:54:41 GMT
View Forum Message <> Reply to Message

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

> This patch applies on top of Paul Menage's container patches (V7) posted at
>
>  http://lkml.org/lkml/2007/2/12/88
>
> It implements a controller within the containers framework for limiting
> memory usage (RSS usage).

It's good to see someone building on someone else's work for once, rather
than everyone going off in different directions.  It makes one hope that we
might actually achieve something at last.

The key part of this patchset is the reclaim algorithm:

> @@ -636,6 +642,15 @@ static unsigned long isolate_lru_pages(u
>
>    list_del(&page->lru);
>    target = src;
> +  /*
> +   * For containers, do not scan the page unless it
> +   * belongs to the container we are reclaiming for
> +   */
> +  if (container && !page_in_container(page, zone, container)) {
> +   scan--;
> +   goto done;
> +  }

Alas, I fear this might have quite bad worst-case behaviour.  One small
container which is under constant memory pressure will churn the
system-wide LRUs like mad, and will consume rather a lot of system time.
So it's a point at which container A can deleteriously affect things which
are running in other containers, which is exactly what we're supposed to

not do.

Subject: Re: [RFC][PATCH][1/4] RSS controller setup
Posted by Andrew Morton on Mon, 19 Feb 2007 08:57:27 GMT
View Forum Message <> Reply to Message

On Mon, 19 Feb 2007 12:20:26 +0530 Balbir Singh <balbir@in.ibm.com> wrote:

>
> This patch sets up the basic controller infrastructure on top of the
> containers infrastructure. Two files are provided for monitoring
> and control  memctlr_usage and memctlr_limit.

The patches use the identifier "memctlr" a lot.  It is hard to remember,
and unpronounceable.  Something like memcontrol or mem_controller or
memory_controller would be more typical.

> ...
>
> + BUG_ON(!mem);
> + if ((buffer = kmalloc(nbytes + 1, GFP_KERNEL)) == 0)
> +  return -ENOMEM;

Please prefer to do

 buffer = kmalloc(nbytes + 1, GFP_KERNEL);
 if (buffer == NULL)
  reutrn -ENOMEM;

ie: avoid the assign-and-test-in-the-same-statement thing.  This affects
the whole patchset.

Also, please don't compare pointers to literal zero like that.  It makes me
get buried it patches to convert it to "NULL".  I think this is a sparse
thing.

> + buffer[nbytes] = 0;
> + if (copy_from_user(buffer, userbuf, nbytes)) {
> +  ret = -EFAULT;
> +  goto out_err;
> + }
> +
> + container_manage_lock();
> + if (container_is_removed(cont)) {
> +  ret = -ENODEV;
> +  goto out_unlock;
> + }

```
> +
> + limit = simple_strtoul(buffer, NULL, 10);
> + /*
> +  * 0 is a valid limit (unlimited resource usage)
> +  */
> + if (!limit && strcmp(buffer, "0"))
> +  goto out_unlock;
> +
> + spin_lock(&mem->lock);
> + mem->counter.limit = limit;
> + spin_unlock(&mem->lock);
```

The patches do this a lot: a single atomic assignment with a pointless-looking lock/unlock around it. It's often the case that this idiom indicates a bug, or needless locking. I think the only case where it makes sense is when there's some other code somewhere which is doing

```
 spin_lock(&mem->lock);
 ...
 use1(mem->counter.limit);
 ...
 use2(mem->counter.limit);
 ...
 spin_unlock(&mem->lock);
```

where use1() and use2() expect the two reads of mem->counter.limit to return the same value.

Is that the case in these patches? If not, we might have a problem in there.

```
> +
> +static ssize_t memctlr_read(struct container *cont, struct cftype *cft,
> +    struct file *file, char __user *userbuf,
> +    size_t nbytes, loff_t *ppos)
> +{
> + unsigned long usage, limit;
> + char usagebuf[64];  /* Move away from stack later */
> + char *s = usagebuf;
> + struct memctlr *mem = memctlr_from_cont(cont);
> +
> + spin_lock(&mem->lock);
> + usage = mem->counter.usage;
> + limit = mem->counter.limit;
> + spin_unlock(&mem->lock);
> +
> + s += sprintf(s, "usage %lu, limit %ld\n", usage, limit);
> + return simple_read_from_buffer(userbuf, nbytes, ppos, usagebuf,
```

> +     s - usagebuf);
> +}

This output is hard to parse and to extend.  I'd suggest either two
separate files, or multi-line output:

usage: %lu kB
limit: %lu kB

and what are the units of these numbers?  Page counts?  If so, please don't
do that: it requires appplications and humans to know the current kernel's
page size.

> +static struct cftype memctlr_usage = {
> + .name = "memctlr_usage",
> + .read = memctlr_read,
> +};
> +
> +static struct cftype memctlr_limit = {
> + .name = "memctlr_limit",
> + .write = memctlr_write,
> +};
> +
> +static int memctlr_populate(struct container_subsys *ss,
> +     struct container *cont)
> +{
> + int rc;
> + if ((rc = container_add_file(cont, &memctlr_usage)) < 0)
> +  return rc;
> + if ((rc = container_add_file(cont, &memctlr_limit)) < 0)

Clean up the first file here?

> +  return rc;
> + return 0;
> +}
> +
> +static struct container_subsys memctlr_subsys = {
> + .name = "memctlr",
> + .create = memctlr_create,
> + .destroy = memctlr_destroy,
> + .populate = memctlr_populate,
> +};
> +
> +int __init memctlr_init(void)
> +{
> + int id;
> +

> + id = container_register_subsys(&memctlr_subsys);
> + printk("Initializing memctlr version %s, id %d\n", version, id);
> + return id < 0 ? id : 0;
> +}
> +
> +module_init(memctlr_init);

---

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

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

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

So it returns zero on success.  OK.

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

>
> + if (!memctlr_mm_init(mm))
> +  goto err;
> +

But here we treat zero as an error?

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

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

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

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

More weird locking here.

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

And here.

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

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

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

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

> +static inline void memctlr_double_unlock(struct memctlr *mem1,
> +       struct memctlr *mem2)
> +{
> + if (mem1 > mem2) {
> +  spin_unlock(&mem2->lock);
> +  spin_unlock(&mem1->lock);
> + } else {
> +  spin_unlock(&mem1->lock);
> +  spin_unlock(&mem2->lock);
> + }
> +}
> +
> ...
>
>   retval = -ENOMEM;
> +
> + if (!memctlr_update_rss(mm, 1, MEMCTLR_CHECK_LIMIT))
> +  goto out;
> +

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

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

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

This one doesn't get checked?

Why does MEMCTLR_DONT_CHECK_LIMIT exist?

---

Subject: Re: [RFC][PATCH][3/4] Add reclaim support
Posted by Andrew Morton on Mon, 19 Feb 2007 08:59:12 GMT
View Forum Message <> Reply to Message

On Mon, 19 Feb 2007 12:20:42 +0530 Balbir Singh <balbir@in.ibm.com> wrote:

>
> This patch reclaims pages from a container when the container limit is hit.
> The executable is oom'ed only when the container it is running in, is overlimit
> and we could not reclaim any pages belonging to the container
>
> A parameter called pushback, controls how much memory is reclaimed when the
> limit is hit. It should be easy to expose this knob to user space, but
> currently it is hard coded to 20% of the total limit of the container.
>
> isolate_lru_pages() has been modified to isolate pages belonging to a
> particular container, so that reclaim code will reclaim only container
> pages. For shared pages, reclaim does not unmap all mappings of the page,
> it only unmaps those mappings that are over their limit. This ensures
> that other containers are not penalized while reclaiming shared pages.
>
> Parallel reclaim per container is not allowed. Each controller has a wait
> queue that ensures that only one task per control is running reclaim on
> that container.
>
>
> ...
>
> --- linux-2.6.20/include/linux/rmap.h~memctlr-reclaim-on-limit 2007-02-18 23:29:14.000000000
+0530
> +++ linux-2.6.20-balbir/include/linux/rmap.h 2007-02-18 23:29:14.000000000 +0530
> @@ -90,7 +90,15 @@ static inline void page_dup_rmap(struct
>  * Called from mm/vmscan.c to handle paging out
>  */
>  int page_referenced(struct page *, int is_locked);
> -int try_to_unmap(struct page *, int ignore_refs);
> +int try_to_unmap(struct page *, int ignore_refs, void *container);
> +#ifdef CONFIG_CONTAINER_MEMCTLR
> +bool page_in_container(struct page *page, struct zone *zone, void *container);
> +#else
> +static inline bool page_in_container(struct page *page, struct zone *zone, void *container)
> +{

> + return true;
> +}
> +#endif /* CONFIG_CONTAINER_MEMCTLR */
>
>  /*
>   * Called from mm/filemap_xip.c to unmap empty zero page
> @@ -118,7 +126,8 @@ int page_mkclean(struct page *);
>  #define anon_vma_link(vma) do {} while (0)
>
>  #define page_referenced(page,l) TestClearPageReferenced(page)
> -#define try_to_unmap(page, refs) SWAP_FAIL
> +#define try_to_unmap(page, refs, container) SWAP_FAIL
> +#define page_in_container(page, zone, container)  true

I spy a compile error.

The static-inline version looks nicer.

>  static inline int page_mkclean(struct page *page)
>  {
> diff -puN include/linux/swap.h~memctlr-reclaim-on-limit include/linux/swap.h
> --- linux-2.6.20/include/linux/swap.h~memctlr-reclaim-on-limit 2007-02-18 23:29:14.000000000
+0530
> +++ linux-2.6.20-balbir/include/linux/swap.h 2007-02-18 23:29:14.000000000 +0530
> @@ -188,6 +188,10 @@ extern void swap_setup(void);
>  /* linux/mm/vmscan.c */
>  extern unsigned long try_to_free_pages(struct zone **, gfp_t);
>  extern unsigned long shrink_all_memory(unsigned long nr_pages);
> +#ifdef CONFIG_CONTAINER_MEMCTLR
> +extern unsigned long memctlr_shrink_mapped_memory(unsigned long nr_pages,
> +       void *container);
> +#endif

Usually one doesn't need to put ifdefs around the declaration like this.
If the function doesn't exist and nobody calls it, we're fine.  If someone
_does_ call it, we'll find out the error at link-time.

>
> +/*
> + * checks if the mm's container and scan control passed container match, if
> + * so, is the container over it's limit. Returns 1 if the container is above
> + * its limit.
> + */
> +int memctlr_mm_overlimit(struct mm_struct *mm, void *sc_cont)
> +{
> + struct container *cont;
> + struct memctlr *mem;
> + long usage, limit;

```
> + int ret = 1;
> +
> + if (!sc_cont)
> +  goto out;
> +
> + read_lock(&mm->container_lock);
> + cont = mm->container;
> +
> + /*
> +   * Regular reclaim, let it proceed as usual
> +   */
> + if (!sc_cont)
> +  goto out;
> +
> + ret = 0;
> + if (cont != sc_cont)
> +  goto out;
> +
> + mem = memctlr_from_cont(cont);
> + usage = atomic_long_read(&mem->counter.usage);
> + limit = atomic_long_read(&mem->counter.limit);
> + if (limit && (usage > limit))
> +  ret = 1;
> +out:
> + read_unlock(&mm->container_lock);
> + return ret;
> +}
```

hm, I wonder how much additional lock traffic all this adds.

```
>  int memctlr_mm_init(struct mm_struct *mm)
>  {
>   mm->counter = kmalloc(sizeof(struct res_counter), GFP_KERNEL);
> @@ -77,6 +125,46 @@ void memctlr_mm_assign_container(struct
>   write_unlock(&mm->container_lock);
>  }
>
> +static int memctlr_check_and_reclaim(struct container *cont, long usage,
> +     long limit)
> +{
> + unsigned long nr_pages = 0;
> + unsigned long nr_reclaimed = 0;
> + int retries = nr_retries;
> + int ret = 1;
> + struct memctlr *mem;
> +
> + mem = memctlr_from_cont(cont);
> + spin_lock(&mem->lock);
```

```
> + while ((retries-- > 0) && limit && (usage > limit)) {
> +  if (mem->reclaim_in_progress) {
> +   spin_unlock(&mem->lock);
> +   wait_event(mem->wq, !mem->reclaim_in_progress);
> +   spin_lock(&mem->lock);
> +  } else {
> +   if (!nr_pages)
> +    nr_pages = (pushback * limit) / 100;
> +   mem->reclaim_in_progress = true;
> +   spin_unlock(&mem->lock);
> +   nr_reclaimed += memctlr_shrink_mapped_memory(nr_pages,
> +         cont);
> +   spin_lock(&mem->lock);
> +   mem->reclaim_in_progress = false;
> +   wake_up_all(&mem->wq);
> +  }
> +  /*
> +    * Resample usage and limit after reclaim
> +    */
> +  usage = atomic_long_read(&mem->counter.usage);
> +  limit = atomic_long_read(&mem->counter.limit);
> + }
> + spin_unlock(&mem->lock);
> +
> + if (limit && (usage > limit))
> +  ret = 0;
> + return ret;
> +}
```

This all looks a bit racy.  And that's common in memory reclaim.  We just have to ensure that when the race happens, we do reasonable things.

I suspect the locking in here could simply be removed.

```
> @@ -66,6 +67,9 @@ struct scan_control {
>   int swappiness;
>
>   int all_unreclaimable;
> +
> + void *container;  /* Used by containers for reclaiming */
> +     /* pages when the limit is exceeded  */
>  };
```

eww.  Why void*?

```
> +#ifdef CONFIG_CONTAINER_MEMCTLR
> +/*
> + * Try to free `nr_pages' of memory, system-wide, and return the number of
```

> + * freed pages.
> + * Modelled after shrink_all_memory()
> + */
> +unsigned long memctlr_shrink_mapped_memory(unsigned long nr_pages, void *container)

80-columns, please.

> +{
> + unsigned long ret = 0;
> + int pass;
> + unsigned long nr_total_scanned = 0;
> +
> + struct scan_control sc = {
> +  .gfp_mask = GFP_KERNEL,
> +  .may_swap = 0,
> +  .swap_cluster_max = nr_pages,
> +  .may_writepage = 1,
> +  .swappiness = vm_swappiness,
> +  .container = container,
> +  .may_swap = 1,
> +  .swappiness = 100,
> + };

swappiness got initialised twice.

> + /*
> +  * We try to shrink LRUs in 3 passes:
> +  * 0 = Reclaim from inactive_list only
> +  * 1 = Reclaim mapped (normal reclaim)
> +  * 2 = 2nd pass of type 1
> +  */
> + for (pass = 0; pass < 3; pass++) {
> +  int prio;
> +
> +  for (prio = DEF_PRIORITY; prio >= 0; prio--) {
> +   unsigned long nr_to_scan = nr_pages - ret;
> +
> +   sc.nr_scanned = 0;
> +   ret += shrink_all_zones(nr_to_scan, prio,
> +      pass, 1, &sc);
> +   if (ret >= nr_pages)
> +    goto out;
> +
> +   nr_total_scanned += sc.nr_scanned;
> +   if (sc.nr_scanned && prio < DEF_PRIORITY - 2)
> +    congestion_wait(WRITE, HZ / 10);
> +  }
> + }

> +out:
> + return ret;
> +}
> +#endif

---

Subject: Re: [RFC][PATCH][0/4] Memory controller (RSS Control)
Posted by Paul Menage on Mon, 19 Feb 2007 09:06:10 GMT
View Forum Message <> Reply to Message

On 2/19/07, Andrew Morton <akpm@linux-foundation.org> wrote:
>
> Alas, I fear this might have quite bad worst-case behaviour. One small
> container which is under constant memory pressure will churn the
> system-wide LRUs like mad, and will consume rather a lot of system time.
> So it's a point at which container A can deleteriously affect things which
> are running in other containers, which is exactly what we're supposed to
> not do.

I think it's OK for a container to consume lots of system time during reclaim, as long as we can account that time to the container involved (i.e. if it's done during direct reclaim rather than by something like kswapd).

Churning the LRU could well be bad though, I agree.

Paul

---

Subject: Re: [RFC][PATCH][0/4] Memory controller (RSS Control)
Posted by Magnus Damm on Mon, 19 Feb 2007 09:16:42 GMT
View Forum Message <> Reply to Message

On 2/19/07, Andrew Morton <akpm@linux-foundation.org> wrote:
> On Mon, 19 Feb 2007 12:20:19 +0530 Balbir Singh <balbir@in.ibm.com> wrote:
>
> > This patch applies on top of Paul Menage's container patches (V7) posted at
> >
> > http://lkml.org/lkml/2007/2/12/88
> >
> > It implements a controller within the containers framework for limiting
> > memory usage (RSS usage).

> The key part of this patchset is the reclaim algorithm:
>
> Alas, I fear this might have quite bad worst-case behaviour. One small
> container which is under constant memory pressure will churn the

> system-wide LRUs like mad, and will consume rather a lot of system time.
> So it's a point at which container A can deleteriously affect things which
> are running in other containers, which is exactly what we're supposed to
> not do.

Nice with a simple memory controller. The downside seems to be that it doesn't scale very well when it comes to reclaim, but maybe that just comes with being simple. Step by step, and maybe this is a good first step?

Ideally I'd like to see unmapped pages handled on a per-container LRU with a fallback to the system-wide LRUs. Shared/mapped pages could be handled using PTE ageing/unmapping instead of page ageing, but that may consume too much resources to be practical.

/ magnus

---

Subject: Re: [RFC][PATCH][1/4] RSS controller setup
Posted by Paul Menage on Mon, 19 Feb 2007 09:18:55 GMT
View Forum Message <> Reply to Message

On 2/19/07, Andrew Morton <akpm@linux-foundation.org> wrote:
>
> This output is hard to parse and to extend.  I'd suggest either two
> separate files, or multi-line output:
>
> usage: %lu kB
> limit: %lu kB

Two separate files would be the container usage model that I envisaged, inherited from the way cpusets does things.

And in this case, it should definitely be the limit in one file, readable and writeable, and the usage in another, probably only readable.

Having to read a file called memctlr_usage to find the current limit sounds wrong.

Hmm, I don't appear to have documented this yet, but I think a good naming scheme for container files is <subsystem>.<whatever> - i.e. these should be memctlr.usage and memctlr.limit. The existing grandfathered Cpusets names violate this, but I'm not sure there's a lot we can do about that.

> > +static int memctlr_populate(struct container_subsys *ss,
> > +                            struct container *cont)

> > +{
> > +	int rc;
> > +	if ((rc = container_add_file(cont, &memctlr_usage)) < 0)
> > +		return rc;
> > +	if ((rc = container_add_file(cont, &memctlr_limit)) < 0)
>
> Clean up the first file here?

Containers don't currently provide an API for a subsystem to clean up
files from a directory - that's done automatically when the directory
is deleted.

I think I'll probably change the API for container_add_file to return
void, but mark an error in the container itself if something goes
wrong - that way rather than all the subsystems having to check for
error, container_populate_dir() can do so at the end of calling all
the subsystems' populate methods.

Paul

---

Subject: Re: [ckrm-tech] [RFC][PATCH][0/4] Memory controller (RSS Control)
Posted by dev on Mon, 19 Feb 2007 09:46:15 GMT
View Forum Message <> Reply to Message

> On 2/19/07, Andrew Morton <akpm@linux-foundation.org> wrote:
>
>>Alas, I fear this might have quite bad worst-case behaviour.  One small
>>container which is under constant memory pressure will churn the
>>system-wide LRUs like mad, and will consume rather a lot of system time.
>>So it's a point at which container A can deleteriously affect things which
>>are running in other containers, which is exactly what we're supposed to
>>not do.
>
>
> I think it's OK for a container to consume lots of system time during
> reclaim, as long as we can account that time to the container involved
> (i.e. if it's done during direct reclaim rather than by something like
> kswapd).
hmm, is it ok to scan 100Gb of RAM for 10MB RAM container?
in UBC patch set we used page beancounters to track containter pages.
This allows to make efficient scan contoler and reclamation.

Thanks,
Kirill

---

Subject: Re: [RFC][PATCH][3/4] Add reclaim support
Posted by KAMEZAWA Hiroyuki on Mon, 19 Feb 2007 09:48:39 GMT
View Forum Message <> Reply to Message

On Mon, 19 Feb 2007 12:20:42 +0530
Balbir Singh <balbir@in.ibm.com> wrote:

> +int memctlr_mm_overlimit(struct mm_struct *mm, void *sc_cont)
> +{
> + struct container *cont;
> + struct memctlr *mem;
> + long usage, limit;
> + int ret = 1;
> +
> + if (!sc_cont)
> + goto out;
> +
> + read_lock(&mm->container_lock);
> + cont = mm->container;

> +out:
> + read_unlock(&mm->container_lock);
> + return ret;
> +}
> +

should be
==
out_and_unlock:
 read_unlock(&mm->container_lock);
out_:
 return ret;

-Kame

Subject: Re: [ckrm-tech] [RFC][PATCH][0/4] Memory controller (RSS Control)
Posted by Paul Menage on Mon, 19 Feb 2007 09:50:22 GMT
View Forum Message <> Reply to Message

On 2/19/07, Kirill Korotaev <dev@sw.ru> wrote:
> >
> > I think it's OK for a container to consume lots of system time during
> > reclaim, as long as we can account that time to the container involved
> > (i.e. if it's done during direct reclaim rather than by something like
> > kswapd).
> hmm, is it ok to scan 100Gb of RAM for 10MB RAM container?

> in UBC patch set we used page beancounters to track containter pages.
> This allows to make efficient scan contoler and reclamation.

I don't mean that we shouldn't go for the most efficient method that's practical. If we can do reclaim without spinning across so much of the LRU, then that's obviously better.

But if the best approach in the general case results in a process in the container spending lots of CPU time trying to do the reclaim, that's probably OK as long as we can account for that time and (once we have a CPU controller) throttle back the container in that case. So then, a container can only hurt itself by thrashing/reclaiming, rather than hurting other containers. (LRU churn notwithstanding ...)

Paul

---

Subject: Re: [RFC][PATCH][0/4] Memory controller (RSS Control)
Posted by Balbir Singh on Mon, 19 Feb 2007 10:00:39 GMT
View Forum Message <> Reply to Message

Andrew Morton wrote:
> On Mon, 19 Feb 2007 12:20:19 +0530 Balbir Singh <balbir@in.ibm.com> wrote:
>
>> This patch applies on top of Paul Menage's container patches (V7) posted at
>>
>>  http://lkml.org/lkml/2007/2/12/88
>>
>> It implements a controller within the containers framework for limiting
>> memory usage (RSS usage).
>
> It's good to see someone building on someone else's work for once, rather
> than everyone going off in different directions.  It makes one hope that we
> might actually achieve something at last.
>

Thanks! It's good to know we are headed in the right direction.

>
> The key part of this patchset is the reclaim algorithm:
>
>> @@ -636,6 +642,15 @@ static unsigned long isolate_lru_pages(u
>>
>>    list_del(&page->lru);
>>    target = src;
>> +  /*
>> +   * For containers, do not scan the page unless it
>> +   * belongs to the container we are reclaiming for

>> +    */
>> +  if (container && !page_in_container(page, zone, container)) {
>> +   scan--;
>> +   goto done;
>> +  }
>
> Alas, I fear this might have quite bad worst-case behaviour.  One small
> container which is under constant memory pressure will churn the
> system-wide LRUs like mad, and will consume rather a lot of system time.
> So it's a point at which container A can deleteriously affect things which
> are running in other containers, which is exactly what we're supposed to
> not do.
>

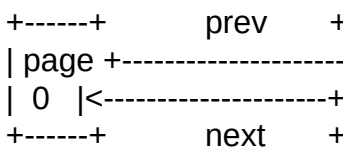
Hmm.. I guess it's space vs time then :-) A CPU controller could
control how much time is spent reclaiming ;)

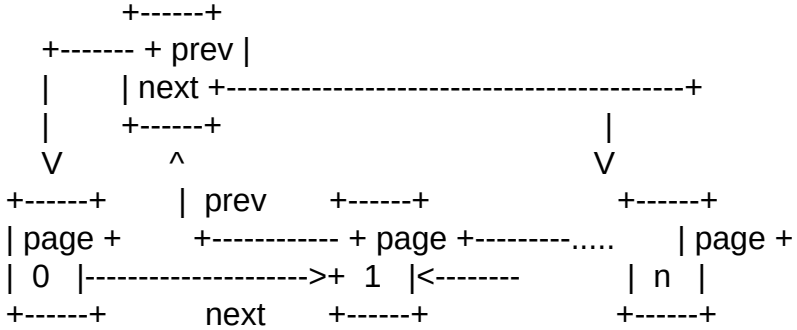
Coming back, I see the problem you mentioned and we have been thinking
of several possible solutions. In my introduction I pointed out

> "Come up with cool page replacement algorithms for containers
> (if possible without any changes to struct page)"

The solutions we have looked at are

1. Overload the LRU list_head in struct page to have a global
   LRU + a per container LRU



Page 1 and n belong to the same container, to get to page 0, you need
two de-references

2. Modify struct page to point to a container and allow each container to
   have a per-container LRU along with the global LRU

For efficiency we need the container LRU and we don't want to split
the global LRU either.

We need to optimize the reclaim path, but I thought of that as a secondary
problem. Once we all agree that the controller looks simple, accounts well
and works. We can/should definitely optimize the reclaim path.

--
 Warm Regards,
 Balbir Singh

---

Subject: Re: [RFC][PATCH][1/4] RSS controller setup
Posted by Balbir Singh on Mon, 19 Feb 2007 10:06:49 GMT
View Forum Message <> Reply to Message

Andrew Morton wrote:
> On Mon, 19 Feb 2007 12:20:26 +0530 Balbir Singh <balbir@in.ibm.com> wrote:
>
>> This patch sets up the basic controller infrastructure on top of the
>> containers infrastructure. Two files are provided for monitoring
>> and control  memctlr_usage and memctlr_limit.
>
> The patches use the identifier "memctlr" a lot.  It is hard to remember,
> and unpronounceable.  Something like memcontrol or mem_controller or
> memory_controller would be more typical.
>

I'll change the name to memory_controller

>> ...
>>
>> + BUG_ON(!mem);
>> + if ((buffer = kmalloc(nbytes + 1, GFP_KERNEL)) == 0)
>> +  return -ENOMEM;
>
> Please prefer to do
>

>  buffer = kmalloc(nbytes + 1, GFP_KERNEL);
>  if (buffer == NULL)
>   reutrn -ENOMEM;
>
> ie: avoid the assign-and-test-in-the-same-statement thing.  This affects
> the whole patchset.
>

I'll fix that

> Also, please don't compare pointers to literal zero like that.  It makes me
> get buried it patches to convert it to "NULL".  I think this is a sparse
> thing.
>

Good point, I'll fix it.

>> + buffer[nbytes] = 0;
>> + if (copy_from_user(buffer, userbuf, nbytes)) {
>> +  ret = -EFAULT;
>> +  goto out_err;
>> + }
>> +
>> + container_manage_lock();
>> + if (container_is_removed(cont)) {
>> +  ret = -ENODEV;
>> +  goto out_unlock;
>> + }
>> +
>> + limit = simple_strtoul(buffer, NULL, 10);
>> + /*
>> +  * 0 is a valid limit (unlimited resource usage)
>> +  */
>> + if (!limit && strcmp(buffer, "0"))
>> +  goto out_unlock;
>> +
>> + spin_lock(&mem->lock);
>> + mem->counter.limit = limit;
>> + spin_unlock(&mem->lock);
>
> The patches do this a lot: a single atomic assignment with a
> pointless-looking lock/unlock around it.  It's often the case that this
> idiom indicates a bug, or needless locking.  I think the only case where it
> makes sense is when there's some other code somewhere which is doing
>
>  spin_lock(&mem->lock);
>  ...
>  use1(mem->counter.limit);

>  ...
>  use2(mem->counter.limit);
>  ...
>  spin_unlock(&mem->lock);
>
> where use1() and use2() expect the two reads of mem->counter.limit to
> return the same value.
>
> Is that the case in these patches?  If not, we might have a problem in
> there.
>

The next set of patches move to atomic values for the limits. That should
fix the locking.

>> +
>> +static ssize_t memctlr_read(struct container *cont, struct cftype *cft,
>> +    struct file *file, char __user *userbuf,
>> +    size_t nbytes, loff_t *ppos)
>> +{
>> + unsigned long usage, limit;
>> + char usagebuf[64];  /* Move away from stack later */
>> + char *s = usagebuf;
>> + struct memctlr *mem = memctlr_from_cont(cont);
>> +
>> + spin_lock(&mem->lock);
>> + usage = mem->counter.usage;
>> + limit = mem->counter.limit;
>> + spin_unlock(&mem->lock);
>> +
>> + s += sprintf(s, "usage %lu, limit %ld\n", usage, limit);
>> + return simple_read_from_buffer(userbuf, nbytes, ppos, usagebuf,
>> +     s - usagebuf);
>> +}
>
> This output is hard to parse and to extend.  I'd suggest either two
> separate files, or multi-line output:
>
> usage: %lu kB
> limit: %lu kB
>
> and what are the units of these numbers?  Page counts?  If so, please don't
> do that: it requires appplications and humans to know the current kernel's
> page size.
>

Yes, this looks much better. I'll move to this format. I get myself lost
in "bc" at times, that should have been a hint.

```
>> +static struct cftype memctlr_usage = {
>> + .name = "memctlr_usage",
>> + .read = memctlr_read,
>> +};
>> +
>> +static struct cftype memctlr_limit = {
>> + .name = "memctlr_limit",
>> + .write = memctlr_write,
>> +};
>> +
>> +static int memctlr_populate(struct container_subsys *ss,
>> +    struct container *cont)
>> +{
>> + int rc;
>> + if ((rc = container_add_file(cont, &memctlr_usage)) < 0)
>> +  return rc;
>> + if ((rc = container_add_file(cont, &memctlr_limit)) < 0)
>
> Clean up the first file here?
>
```

I used cpuset_populate() as an example to code this one up.
I don't think there is an easy way in containers to clean up
files. I'll double check

```
>> +  return rc;
>> + return 0;
>> +}
>> +
>> +static struct container_subsys memctlr_subsys = {
>> + .name = "memctlr",
>> + .create = memctlr_create,
>> + .destroy = memctlr_destroy,
>> + .populate = memctlr_populate,
>> +};
>> +
>> +int __init memctlr_init(void)
>> +{
>> + int id;
>> +
>> + id = container_register_subsys(&memctlr_subsys);
>> + printk("Initializing memctlr version %s, id %d\n", version, id);
>> + return id < 0 ? id : 0;
>> +}
>> +
>> +module_init(memctlr_init);
>
```

Thanks for the detailed review,

--
Warm Regards,
Balbir Singh

---

Subject: Re: [ckrm-tech] [RFC][PATCH][0/4] Memory controller (RSS Control)
Posted by Balbir Singh on Mon, 19 Feb 2007 10:24:07 GMT
View Forum Message <> Reply to Message

Kirill Korotaev wrote:
>> On 2/19/07, Andrew Morton <akpm@linux-foundation.org> wrote:
>>
>>> Alas, I fear this might have quite bad worst-case behaviour.  One small
>>> container which is under constant memory pressure will churn the
>>> system-wide LRUs like mad, and will consume rather a lot of system time.
>>> So it's a point at which container A can deleteriously affect things which
>>> are running in other containers, which is exactly what we're supposed to
>>> not do.
>>
>> I think it's OK for a container to consume lots of system time during
>> reclaim, as long as we can account that time to the container involved
>> (i.e. if it's done during direct reclaim rather than by something like
>> kswapd).
> hmm, is it ok to scan 100Gb of RAM for 10MB RAM container?
> in UBC patch set we used page beancounters to track containter pages.
> This allows to make efficient scan contoler and reclamation.
>
> Thanks,
> Kirill

Hi, Kirill,

Yes, that's a problem, but I think it's a problem that can be solved
in steps. First step, add reclaim. Second step, optimize reclaim.

--
Warm Regards,
Balbir Singh

---

Subject: Re: [RFC][PATCH][0/4] Memory controller (RSS Control)
Posted by Balbir Singh on Mon, 19 Feb 2007 10:39:26 GMT
View Forum Message <> Reply to Message

Paul Menage wrote:
> On 2/19/07, Andrew Morton <akpm@linux-foundation.org> wrote:
>>
>> Alas, I fear this might have quite bad worst-case behaviour.  One small
>> container which is under constant memory pressure will churn the
>> system-wide LRUs like mad, and will consume rather a lot of system time.
>> So it's a point at which container A can deleteriously affect things
>> which
>> are running in other containers, which is exactly what we're supposed to
>> not do.
>
> I think it's OK for a container to consume lots of system time during
> reclaim, as long as we can account that time to the container involved
> (i.e. if it's done during direct reclaim rather than by something like
> kswapd).
>
> Churning the LRU could well be bad though, I agree.
>

I completely agree with you on reclaim consuming time.

Churning the LRU can be avoided by the means I mentioned before

1. Add a container pointer (per page struct), it is also
   useful for the page cache controller
2. Check if the page belongs to a particular container before
   the list_del(&page->lru), so that those pages can be skipped.
3. Use a double LRU list by overloading the lru list_head of
   struct page.

> Paul
>

--
Warm Regards,
Balbir Singh

---

Subject: Re: [RFC][PATCH][0/4] Memory controller (RSS Control)
Posted by Balbir Singh on Mon, 19 Feb 2007 10:45:01 GMT
View Forum Message <> Reply to Message

Magnus Damm wrote:
> On 2/19/07, Andrew Morton <akpm@linux-foundation.org> wrote:
>> On Mon, 19 Feb 2007 12:20:19 +0530 Balbir Singh <balbir@in.ibm.com>
>> wrote:
>>

>> > This patch applies on top of Paul Menage's container patches (V7)
>> posted at
>> >
>> >       http://lkml.org/lkml/2007/2/12/88
>> >
>> > It implements a controller within the containers framework for limiting
>> > memory usage (RSS usage).
>
>> The key part of this patchset is the reclaim algorithm:
>>
>> Alas, I fear this might have quite bad worst-case behaviour.  One small
>> container which is under constant memory pressure will churn the
>> system-wide LRUs like mad, and will consume rather a lot of system time.
>> So it's a point at which container A can deleteriously affect things
>> which
>> are running in other containers, which is exactly what we're supposed to
>> not do.
>
> Nice with a simple memory controller. The downside seems to be that it
> doesn't scale very well when it comes to reclaim, but maybe that just
> comes with being simple. Step by step, and maybe this is a good first
> step?
>

Thanks, I totally agree.

> Ideally I'd like to see unmapped pages handled on a per-container LRU
> with a fallback to the system-wide LRUs. Shared/mapped pages could be
> handled using PTE ageing/unmapping instead of page ageing, but that
> may consume too much resources to be practical.
>
> / magnus

Keeping unmapped pages per container sounds interesting. I am not quite
sure what PTE ageing, will it look it up.

--
Warm Regards,
Balbir Singh

---

Subject: Re: [RFC][PATCH][3/4] Add reclaim support
Posted by Balbir Singh on Mon, 19 Feb 2007 10:50:53 GMT
View Forum Message <> Reply to Message

Andrew Morton wrote:
> On Mon, 19 Feb 2007 12:20:42 +0530 Balbir Singh <balbir@in.ibm.com> wrote:

>
>> This patch reclaims pages from a container when the container limit is hit.
>> The executable is oom'ed only when the container it is running in, is overlimit
>> and we could not reclaim any pages belonging to the container
>>
>> A parameter called pushback, controls how much memory is reclaimed when the
>> limit is hit. It should be easy to expose this knob to user space, but
>> currently it is hard coded to 20% of the total limit of the container.
>>
>> isolate_lru_pages() has been modified to isolate pages belonging to a
>> particular container, so that reclaim code will reclaim only container
>> pages. For shared pages, reclaim does not unmap all mappings of the page,
>> it only unmaps those mappings that are over their limit. This ensures
>> that other containers are not penalized while reclaiming shared pages.
>>
>> Parallel reclaim per container is not allowed. Each controller has a wait
>> queue that ensures that only one task per control is running reclaim on
>> that container.
>>
>>
>> ...
>>
>> --- linux-2.6.20/include/linux/rmap.h~memctlr-reclaim-on-limit 2007-02-18 23:29:14.000000000
+0530
>> +++ linux-2.6.20-balbir/include/linux/rmap.h 2007-02-18 23:29:14.000000000 +0530
>> @@ -90,7 +90,15 @@ static inline void page_dup_rmap(struct
>>  * Called from mm/vmscan.c to handle paging out
>>  */
>> int page_referenced(struct page *, int is_locked);
>> -int try_to_unmap(struct page *, int ignore_refs);
>> +int try_to_unmap(struct page *, int ignore_refs, void *container);
>> +#ifdef CONFIG_CONTAINER_MEMCTLR
>> +bool page_in_container(struct page *page, struct zone *zone, void *container);
>> +#else
>> +static inline bool page_in_container(struct page *page, struct zone *zone, void *container)
>> +{
>> + return true;
>> +}
>> +#endif /* CONFIG_CONTAINER_MEMCTLR */
>>
>> /*
>>  * Called from mm/filemap_xip.c to unmap empty zero page
>> @@ -118,7 +126,8 @@ int page_mkclean(struct page *);
>> #define anon_vma_link(vma) do {} while (0)
>>
>> #define page_referenced(page,l) TestClearPageReferenced(page)
>> -#define try_to_unmap(page, refs) SWAP_FAIL
>> +#define try_to_unmap(page, refs, container) SWAP_FAIL

>> +#define page_in_container(page, zone, container)  true
>
> I spy a compile error.
>
> The static-inline version looks nicer.
>

I will compile with the feature turned off and double check. I'll
also convert it to a static inline function.

>>  static inline int page_mkclean(struct page *page)
>>  {
>> diff -puN include/linux/swap.h~memctlr-reclaim-on-limit include/linux/swap.h
>> --- linux-2.6.20/include/linux/swap.h~memctlr-reclaim-on-limit 2007-02-18 23:29:14.000000000
+0530
>> +++ linux-2.6.20-balbir/include/linux/swap.h 2007-02-18 23:29:14.000000000 +0530
>> @@ -188,6 +188,10 @@ extern void swap_setup(void);
>>  /* linux/mm/vmscan.c */
>>  extern unsigned long try_to_free_pages(struct zone **, gfp_t);
>>  extern unsigned long shrink_all_memory(unsigned long nr_pages);
>> +#ifdef CONFIG_CONTAINER_MEMCTLR
>> +extern unsigned long memctlr_shrink_mapped_memory(unsigned long nr_pages,
>> +       void *container);
>> +#endif
>
> Usually one doesn't need to put ifdefs around the declaration like this.
> If the function doesn't exist and nobody calls it, we're fine.  If someone
> _does_ call it, we'll find out the error at link-time.
>

Sure, sounds good. I'll get rid of the #ifdefs.

>>
>> +/*
>> + * checks if the mm's container and scan control passed container match, if
>> + * so, is the container over it's limit. Returns 1 if the container is above
>> + * its limit.
>> + */
>> +int memctlr_mm_overlimit(struct mm_struct *mm, void *sc_cont)
>> +{
>> + struct container *cont;
>> + struct memctlr *mem;
>> + long usage, limit;
>> + int ret = 1;
>> +
>> + if (!sc_cont)

```
>> +  goto out;
>> +
>> + read_lock(&mm->container_lock);
>> + cont = mm->container;
>> +
>> + /*
>> +   * Regular reclaim, let it proceed as usual
>> +   */
>> + if (!sc_cont)
>> +  goto out;
>> +
>> + ret = 0;
>> + if (cont != sc_cont)
>> +  goto out;
>> +
>> + mem = memctlr_from_cont(cont);
>> + usage = atomic_long_read(&mem->counter.usage);
>> + limit = atomic_long_read(&mem->counter.limit);
>> + if (limit && (usage > limit))
>> +  ret = 1;
>> +out:
>> + read_unlock(&mm->container_lock);
>> + return ret;
>> +}
>
> hm, I wonder how much additional lock traffic all this adds.
>
```

It's a read_lock() and most of the locks are read_locks
which allow for concurrent access, until the container
changes or goes away

```
>>  int memctlr_mm_init(struct mm_struct *mm)
>>  {
>>   mm->counter = kmalloc(sizeof(struct res_counter), GFP_KERNEL);
>> @@ -77,6 +125,46 @@ void memctlr_mm_assign_container(struct
>>   write_unlock(&mm->container_lock);
>>  }
>>
>> +static int memctlr_check_and_reclaim(struct container *cont, long usage,
>> +     long limit)
>> +{
>> + unsigned long nr_pages = 0;
>> + unsigned long nr_reclaimed = 0;
>> + int retries = nr_retries;
>> + int ret = 1;
>> + struct memctlr *mem;
>> +
```

```
>> + mem = memctlr_from_cont(cont);
>> + spin_lock(&mem->lock);
>> + while ((retries-- > 0) && limit && (usage > limit)) {
>> +  if (mem->reclaim_in_progress) {
>> +   spin_unlock(&mem->lock);
>> +   wait_event(mem->wq, !mem->reclaim_in_progress);
>> +   spin_lock(&mem->lock);
>> +  } else {
>> +   if (!nr_pages)
>> +    nr_pages = (pushback * limit) / 100;
>> +   mem->reclaim_in_progress = true;
>> +   spin_unlock(&mem->lock);
>> +   nr_reclaimed += memctlr_shrink_mapped_memory(nr_pages,
>> +         cont);
>> +   spin_lock(&mem->lock);
>> +   mem->reclaim_in_progress = false;
>> +   wake_up_all(&mem->wq);
>> +  }
>> +  /*
>> +    * Resample usage and limit after reclaim
>> +    */
>> +  usage = atomic_long_read(&mem->counter.usage);
>> +  limit = atomic_long_read(&mem->counter.limit);
>> + }
>> + spin_unlock(&mem->lock);
>> +
>> + if (limit && (usage > limit))
>> +  ret = 0;
>> + return ret;
>> +}
>
> This all looks a bit racy.  And that's common in memory reclaim.  We just
> have to ensure that when the race happens, we do reasonable things.
>
> I suspect the locking in here could simply be removed.
>
```

The locking is mostly to ensure that tasks belonging to the same container
see a consistent value of reclaim_in_progress. I'll see if the locking
can be simplified or simply removed.

```
>> @@ -66,6 +67,9 @@ struct scan_control {
>>   int swappiness;
>>
>>   int all_unreclaimable;
>> +
>> + void *container;  /* Used by containers for reclaiming */
>> +     /* pages when the limit is exceeded  */
```

>>  };
>
> eww.  Why void*?
>

I did not want to expose struct container in mm/vmscan.c. An additional
thought was that no matter what container goes in the field would be
useful for reclaim.

>> +#ifdef CONFIG_CONTAINER_MEMCTLR
>> +/*
>> + * Try to free `nr_pages' of memory, system-wide, and return the number of
>> + * freed pages.
>> + * Modelled after shrink_all_memory()
>> + */
>> +unsigned long memctlr_shrink_mapped_memory(unsigned long nr_pages, void *container)
>
> 80-columns, please.
>

I'll fix this.

>> +{
>> + unsigned long ret = 0;
>> + int pass;
>> + unsigned long nr_total_scanned = 0;
>> +
>> + struct scan_control sc = {
>> +  .gfp_mask = GFP_KERNEL,
>> +  .may_swap = 0,
>> +  .swap_cluster_max = nr_pages,
>> +  .may_writepage = 1,
>> +  .swappiness = vm_swappiness,
>> +  .container = container,
>> +  .may_swap = 1,
>> +  .swappiness = 100,
>> + };
>
> swappiness got initialised twice.
>

I should have caught that earlier. Thanks for spotting this.
I'll fix it.

>> + /*
>> +  * We try to shrink LRUs in 3 passes:
>> +  * 0 = Reclaim from inactive_list only
>> +  * 1 = Reclaim mapped (normal reclaim)

```
>> +  * 2 = 2nd pass of type 1
>> +  */
>> + for (pass = 0; pass < 3; pass++) {
>> +  int prio;
>> +
>> +  for (prio = DEF_PRIORITY; prio >= 0; prio--) {
>> +   unsigned long nr_to_scan = nr_pages - ret;
>> +
>> +   sc.nr_scanned = 0;
>> +   ret += shrink_all_zones(nr_to_scan, prio,
>> +      pass, 1, &sc);
>> +   if (ret >= nr_pages)
>> +    goto out;
>> +
>> +   nr_total_scanned += sc.nr_scanned;
>> +   if (sc.nr_scanned && prio < DEF_PRIORITY - 2)
>> +    congestion_wait(WRITE, HZ / 10);
>> +  }
>> + }
>> +out:
>> + return ret;
>> +}
>> +#endif
>
```

--
Warm Regards,
Balbir Singh

---

Subject: Re: [RFC][PATCH][3/4] Add reclaim support
Posted by Balbir Singh on Mon, 19 Feb 2007 10:52:35 GMT
View Forum Message <> Reply to Message

KAMEZAWA Hiroyuki wrote:
> On Mon, 19 Feb 2007 12:20:42 +0530
> Balbir Singh <balbir@in.ibm.com> wrote:
>
```
>> +int memctlr_mm_overlimit(struct mm_struct *mm, void *sc_cont)
>> +{
>> + struct container *cont;
>> + struct memctlr *mem;
>> + long usage, limit;
>> + int ret = 1;
>> +
>> + if (!sc_cont)
>> +  goto out;
```

>> +
>> + read_lock(&mm->container_lock);
>> + cont = mm->container;
>
>> +out:
>> + read_unlock(&mm->container_lock);
>> + return ret;
>> +}
>> +
>
> should be
> ==
> out_and_unlock:
>  read_unlock(&mm->container_lock);
> out_:
>  return ret;
>


Thanks, that's a much convention!

>
> -Kame
>


--
Warm Regards,
Balbir Singh

---

Subject: Re: [RFC][PATCH][3/4] Add reclaim support
Posted by Andrew Morton on Mon, 19 Feb 2007 11:10:17 GMT
View Forum Message <> Reply to Message

On Mon, 19 Feb 2007 16:20:53 +0530 Balbir Singh <balbir@in.ibm.com> wrote:

> >> + * so, is the container over it's limit. Returns 1 if the container is above
> >> + * its limit.
> >> + */
> >> +int memctlr_mm_overlimit(struct mm_struct *mm, void *sc_cont)
> >> +{
> >> + struct container *cont;
> >> + struct memctlr *mem;
> >> + long usage, limit;
> >> + int ret = 1;
> >> +
> >> + if (!sc_cont)

> >> +  goto out;
> >> +
> >> + read_lock(&mm->container_lock);
> >> + cont = mm->container;
> >> +
> >> + /*
> >> +   * Regular reclaim, let it proceed as usual
> >> +   */
> >> + if (!sc_cont)
> >> +  goto out;
> >> +
> >> + ret = 0;
> >> + if (cont != sc_cont)
> >> +  goto out;
> >> +
> >> + mem = memctlr_from_cont(cont);
> >> + usage = atomic_long_read(&mem->counter.usage);
> >> + limit = atomic_long_read(&mem->counter.limit);
> >> + if (limit && (usage > limit))
> >> +  ret = 1;
> >> +out:
> >> + read_unlock(&mm->container_lock);
> >> + return ret;
> >> +}
> >
> > hm, I wonder how much additional lock traffic all this adds.
> >
>
> It's a read_lock() and most of the locks are read_locks
> which allow for concurrent access, until the container
> changes or goes away

read_lock isn't free, and I suspect we're calling this function pretty
often (every pagefault?) It'll be measurable on some workloads, on some
hardware.

It probably won't be terribly bad because each lock-taking is associated
with a clear_page().  But still, if there's any possibility of lightening
the locking up, now is the time to think about it.

> >> @@ -66,6 +67,9 @@ struct scan_control {
> >>   int swappiness;
> >>
> >>   int all_unreclaimable;
> >> +
> >> + void *container;  /* Used by containers for reclaiming */
> >> +     /* pages when the limit is exceeded  */
> >>  };

> >
> > eww.  Why void*?
> >
>
> I did not want to expose struct container in mm/vmscan.c.

It's already there, via rmap.h

> An additional
> thought was that no matter what container goes in the field would be
> useful for reclaim.

Am having trouble parsing that sentence ;)

---

Subject: Re: [RFC][PATCH][1/4] RSS controller setup
Posted by Balbir Singh on Mon, 19 Feb 2007 11:13:24 GMT
View Forum Message <> Reply to Message

Paul Menage wrote:
> On 2/19/07, Andrew Morton <akpm@linux-foundation.org> wrote:
>>
>> This output is hard to parse and to extend.  I'd suggest either two
>> separate files, or multi-line output:
>>
>> usage: %lu kB
>> limit: %lu kB
>
> Two separate files would be the container usage model that I
> envisaged, inherited from the way cpusets does things.
>
> And in this case, it should definitely be the limit in one file,
> readable and writeable, and the usage in another, probably only
> readable.
>
> Having to read a file called memctlr_usage to find the current limit
> sounds wrong.
>

That sound right, I'll fix this.

> Hmm, I don't appear to have documented this yet, but I think a good
> naming scheme for container files is <subsystem>.<whatever> - i.e.
> these should be memctlr.usage and memctlr.limit. The existing
> grandfathered Cpusets names violate this, but I'm not sure there's a
> lot we can do about that.
>

Why <subsystem>.<whatever>, dots are harder to parse using regular
expressions and sound DOS'ish. I'd prefer "_" to separate the
subsystem and whatever :-)

>> > +static int memctlr_populate(struct container_subsys *ss,
>> > +                            struct container *cont)
>> > +{
>> > +    int rc;
>> > +    if ((rc = container_add_file(cont, &memctlr_usage)) < 0)
>> > +            return rc;
>> > +    if ((rc = container_add_file(cont, &memctlr_limit)) < 0)
>>
>> Clean up the first file here?
>
> Containers don't currently provide an API for a subsystem to clean up
> files from a directory - that's done automatically when the directory
> is deleted.
>
> I think I'll probably change the API for container_add_file to return
> void, but mark an error in the container itself if something goes
> wrong - that way rather than all the subsystems having to check for
> error, container_populate_dir() can do so at the end of calling all
> the subsystems' populate methods.
>

It should be easy to add container_remove_file() instead of marking
an error.

> Paul

--
Warm Regards,
Balbir Singh

---

Subject: Re: [RFC][PATCH][3/4] Add reclaim support
Posted by Balbir Singh on Mon, 19 Feb 2007 11:16:33 GMT
View Forum Message <> Reply to Message

Andrew Morton wrote:
> On Mon, 19 Feb 2007 16:20:53 +0530 Balbir Singh <balbir@in.ibm.com> wrote:
>
>>>> + * so, is the container over it's limit. Returns 1 if the container is above
>>>> + * its limit.
>>>> + */
>>>> +int memctlr_mm_overlimit(struct mm_struct *mm, void *sc_cont)
>>>> +{

>>>> + struct container *cont;
>>>> + struct memctlr *mem;
>>>> + long usage, limit;
>>>> + int ret = 1;
>>>> +
>>>> + if (!sc_cont)
>>>> +  goto out;
>>>> +
>>>> + read_lock(&mm->container_lock);
>>>> + cont = mm->container;
>>>> +
>>>> + /*
>>>> +   * Regular reclaim, let it proceed as usual
>>>> +   */
>>>> + if (!sc_cont)
>>>> +  goto out;
>>>> +
>>>> + ret = 0;
>>>> + if (cont != sc_cont)
>>>> +  goto out;
>>>> +
>>>> + mem = memctlr_from_cont(cont);
>>>> + usage = atomic_long_read(&mem->counter.usage);
>>>> + limit = atomic_long_read(&mem->counter.limit);
>>>> + if (limit && (usage > limit))
>>>> +  ret = 1;
>>>> +out:
>>>> + read_unlock(&mm->container_lock);
>>>> + return ret;
>>>> +}
>>> hm, I wonder how much additional lock traffic all this adds.
>>>
>> It's a read_lock() and most of the locks are read_locks
>> which allow for concurrent access, until the container
>> changes or goes away
>
> read_lock isn't free, and I suspect we're calling this function pretty
> often (every pagefault?) It'll be measurable on some workloads, on some
> hardware.
>
> It probably won't be terribly bad because each lock-taking is associated
> with a clear_page().  But still, if there's any possibility of lightening
> the locking up, now is the time to think about it.
>

Yes, good point. I'll revisit to see if barriers can replace the locking
or if the locking is required at all?

>>>> @@ -66,6 +67,9 @@ struct scan_control {
>>>>  int swappiness;
>>>>
>>>>  int all_unreclaimable;
>>>> +
>>>> + void *container;  /* Used by containers for reclaiming */
>>>> +     /* pages when the limit is exceeded  */
>>>>  };
>>> eww.  Why void*?
>>>
>> I did not want to expose struct container in mm/vmscan.c.
>
> It's already there, via rmap.h
>

Yes, true

>> An additional
>> thought was that no matter what container goes in the field would be
>> useful for reclaim.
>
> Am having trouble parsing that sentence ;)
>
>

The thought was that irrespective of the infrastructure that goes in
having an entry for reclaim in scan_control would be useful. I guess
the name exposes what the type tries to hide :-)

--
 Warm Regards,
 Balbir Singh

---

Subject: Re: [RFC][PATCH][0/4] Memory controller (RSS Control)
Posted by Magnus Damm on Mon, 19 Feb 2007 11:56:06 GMT
View Forum Message <> Reply to Message

On 2/19/07, Balbir Singh <balbir@in.ibm.com> wrote:
> Magnus Damm wrote:
> > On 2/19/07, Andrew Morton <akpm@linux-foundation.org> wrote:
> >> On Mon, 19 Feb 2007 12:20:19 +0530 Balbir Singh <balbir@in.ibm.com>
> >> wrote:
> >>
> >> > This patch applies on top of Paul Menage's container patches (V7)
> >> posted at
> >> >
> >> >       http://lkml.org/lkml/2007/2/12/88

> >> >
> >> > It implements a controller within the containers framework for limiting
> >> > memory usage (RSS usage).
> >
> >> The key part of this patchset is the reclaim algorithm:
> >>
> >> Alas, I fear this might have quite bad worst-case behaviour.  One small
> >> container which is under constant memory pressure will churn the
> >> system-wide LRUs like mad, and will consume rather a lot of system time.
> >> So it's a point at which container A can deleteriously affect things
> >> which
> >> are running in other containers, which is exactly what we're supposed to
> >> not do.
> >
> > Nice with a simple memory controller. The downside seems to be that it
> > doesn't scale very well when it comes to reclaim, but maybe that just
> > comes with being simple. Step by step, and maybe this is a good first
> > step?
> >
>
> Thanks, I totally agree.
>
> > Ideally I'd like to see unmapped pages handled on a per-container LRU
> > with a fallback to the system-wide LRUs. Shared/mapped pages could be
> > handled using PTE ageing/unmapping instead of page ageing, but that
> > may consume too much resources to be practical.
> >
> > / magnus
>
> Keeping unmapped pages per container sounds interesting. I am not quite
> sure what PTE ageing, will it look it up.

You will most likely have no luck looking it up, so here is what I
mean by PTE ageing:

The most common unit for memory resource control seems to be physical
pages. Keeping track of pages is simple in the case of a single user
per page, but for shared pages tracking the owner becomes more
complex.

I consider unmapped pages to only have a single user at a time, so the
unit for unmapped memory resource control is physical pages. Apart
from implementation details such as fun with struct page and
scalability, handling this case is not so complicated.

Mapped or shared pages should be handled in a different way IMO. PTEs
should be used instead of using physical pages as unit for resource
control and reclaim. For the user this looks pretty much the same as

physical pages, apart for memory overcommit.

So instead of using a global page reclaim policy and reserving physical pages per container I propose that resource controlled shared pages should be handled using a PTE replacement policy. This policy is used to keep the most active PTEs in the container backed by physical pages. Inactive PTEs gets unmapped in favour over newer PTEs.

One way to implement this could be by populating the address space of resource controlled processes with multiple smaller LRU2Qs. The compact data structure that I have in mind is basically an array of 256 bytes, one byte per PTE. Associated with this data strucuture are start indexes and lengths for two lists. The indexes are used in a FAT-type of chain to form single linked lists. So we create active and inactive list here - and we move PTEs between the lists when we check the young bits from the page reclaim and when we apply memory pressure. Unmapping is done through the normal page reclaimer but using information from the PTE LRUs.

In my mind this should lead to more fair resource control of mapped pages, but if it is possible to implement with low overhead, that's another question. =)

Thanks for listening.

/ magnus

---

Subject: Re: [RFC][PATCH][0/4] Memory controller (RSS Control)
Posted by Balbir Singh on Mon, 19 Feb 2007 14:07:07 GMT
View Forum Message <> Reply to Message

Magnus Damm wrote:
> On 2/19/07, Balbir Singh <balbir@in.ibm.com> wrote:
>> Magnus Damm wrote:
>> > On 2/19/07, Andrew Morton <akpm@linux-foundation.org> wrote:
>> >> On Mon, 19 Feb 2007 12:20:19 +0530 Balbir Singh <balbir@in.ibm.com>
>> >> wrote:
>> >>
>> >> > This patch applies on top of Paul Menage's container patches (V7)
>> >> posted at
>> >> >
>> >> >       http://lkml.org/lkml/2007/2/12/88
>> >> >
>> >> > It implements a controller within the containers framework for
>> limiting
>> >> > memory usage (RSS usage).
>> >>

>> >> The key part of this patchset is the reclaim algorithm:
>> >>
>> >> Alas, I fear this might have quite bad worst-case behaviour.  One
>> small
>> >> container which is under constant memory pressure will churn the
>> >> system-wide LRUs like mad, and will consume rather a lot of system
>> time.
>> >> So it's a point at which container A can deleteriously affect things
>> >> which
>> >> are running in other containers, which is exactly what we're
>> supposed to
>> >> not do.
>> >
>> > Nice with a simple memory controller. The downside seems to be that it
>> > doesn't scale very well when it comes to reclaim, but maybe that just
>> > comes with being simple. Step by step, and maybe this is a good first
>> > step?
>> >
>>
>> Thanks, I totally agree.
>>
>> > Ideally I'd like to see unmapped pages handled on a per-container LRU
>> > with a fallback to the system-wide LRUs. Shared/mapped pages could be
>> > handled using PTE ageing/unmapping instead of page ageing, but that
>> > may consume too much resources to be practical.
>> >
>> > / magnus
>>
>> Keeping unmapped pages per container sounds interesting. I am not quite
>> sure what PTE ageing, will it look it up.
>
> You will most likely have no luck looking it up, so here is what I
> mean by PTE ageing:
>
> The most common unit for memory resource control seems to be physical
> pages. Keeping track of pages is simple in the case of a single user
> per page, but for shared pages tracking the owner becomes more
> complex.
>
> I consider unmapped pages to only have a single user at a time, so the
> unit for unmapped memory resource control is physical pages. Apart
> from implementation details such as fun with struct page and
> scalability, handling this case is not so complicated.
>
> Mapped or shared pages should be handled in a different way IMO. PTEs
> should be used instead of using physical pages as unit for resource
> control and reclaim. For the user this looks pretty much the same as
> physical pages, apart for memory overcommit.

>
> So instead of using a global page reclaim policy and reserving
> physical pages per container I propose that resource controlled shared
> pages should be handled using a PTE replacement policy. This policy is
> used to keep the most active PTEs in the container backed by physical
> pages. Inactive PTEs gets unmapped in favour over newer PTEs.
>
> One way to implement this could be by populating the address space of
> resource controlled processes with multiple smaller LRU2Qs. The
> compact data structure that I have in mind is basically an array of
> 256 bytes, one byte per PTE. Associated with this data strucuture are
> start indexes and lengths for two lists. The indexes are used in a
> FAT-type of chain to form single linked lists. So we create active and
> inactive list here - and we move PTEs between the lists when we check
> the young bits from the page reclaim and when we apply memory
> pressure. Unmapping is done through the normal page reclaimer but
> using information from the PTE LRUs.
>
> In my mind this should lead to more fair resource control of mapped
> pages, but if it is possible to implement with low overhead, that's
> another question. =)
>
> Thanks for listening.
>
> / magnus
>

Thanks for explaining PTE aging.

--
Warm Regards,
Balbir Singh

---

Subject: Re: [RFC][PATCH][1/4] RSS controller setup
Posted by Matt Helsley on Mon, 19 Feb 2007 19:43:46 GMT
View Forum Message <> Reply to Message

On Mon, 2007-02-19 at 16:43 +0530, Balbir Singh wrote:
> Paul Menage wrote:
> > On 2/19/07, Andrew Morton <akpm@linux-foundation.org> wrote:

<snip>

> > Hmm, I don't appear to have documented this yet, but I think a good
> > naming scheme for container files is <subsystem>.<whatever> - i.e.
> > these should be memctlr.usage and memctlr.limit. The existing
> > grandfathered Cpusets names violate this, but I'm not sure there's a

> > lot we can do about that.
> >
>
> Why <subsystem>.<whatever>, dots are harder to parse using regular
> expressions and sound DOS'ish. I'd prefer "_" to separate the
> subsystem and whatever :-)

"_" is useful for names with "spaces". Names like mem_controller. "."
seems reasonable despite its regex nastyness. Alternatively there's
always ":".

<snip>

Cheers,
 -Matt Helsley

---