Subject: [RFC][PATCH][0/4] Memory controller (RSS Control) (
Posted by Balbir Singh on Sat, 24 Feb 2007 14:45:03 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

This is version 2 of the patch, version 1 was posted at
http://lkml.org/lkml/2007/2/19/10

I have tried to incorporate all comments, more details can be found
in the changelog's of induvidual patches. Any remaining mistakes are
all my fault.

The next question could be why release version 2?

1. It serves a decision point to decide if we should move to a per-container
   LRU list. Walking through the global LRU is slow, in this patchset I've
   tried to address the LRU churning issue. The patch
   memcontrol-reclaim-on-limit has more details
2. I;ve included fixes for several of the comments/issues raised in version 1

Steps to use the controller
---
0. Download the patches, apply the patches
1. Turn on CONFIG_CONTAINER_MEMCONTROL in kernel config, build the kernel
   and boot into the new kernel
2. mount -t container container -o memcontrol /<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) > memcontrol_limit
6. cat memcontrol_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

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.
Tested the patches on UML and PowerPC. On UML tried the patches with the
config enabled and disabled (sanity check) and with containers enabled
but the memory controller disabled.

TODO's and improvement areas
---
1. Come up with cool page replacement algorithms for containers - still holds
   good (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
---
memcontrol-setup.patch
memcontrol-acct.patch
memcontrol-reclaim-on-limit.patch
memcontrol-doc.patch

--
Warm Regards,
Balbir Singh

Subject: [RFC][PATCH][1/4] RSS controller setup (
Posted by Balbir Singh on Sat, 24 Feb 2007 14:45:13 GMT
View Forum Message <> Reply to Message

Changelog

1. Change the name from memctlr to memcontrol
2. Coding style changes, call the API and then check return value (for kmalloc).
3. Change the output format, to print sizes in both pages and kB
4. Split the usage and limit files to be independent (cat memcontrol_usage

no longer prints the limit)

TODO's

1. Implement error handling mechansim for handling container_add_file()
   failures (this would depend on the containers code).

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

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

memcontrol_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/memcontrol.h | 33 +++++++
 init/Kconfig | 7 +
 mm/Makefile | 1
 mm/memcontrol.c | 193 ++
 4 files changed, 234 insertions(+)

diff -puN /dev/null include/linux/memcontrol.h
--- /dev/null 2007-02-02 22:51:23.000000000 +0530
+++ linux-2.6.20-balbir/include/linux/memcontrol.h 2007-02-24 19:39:03.000000000 +0530
@@ -0,0 +1,33 @@
+/*
+ * memcontrol.h - Memory Controller for containers
+ *
+ * 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.
+ *
+ * You should have received a copy of the GNU General Public License
+ * along with this program; if not, write to the Free Software
+ * Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
+ *
+ * © Copyright IBM Corporation, 2006-2007
+ *

```
+ * Author: Balbir Singh <balbir@in.ibm.com>
+ *
+ */
+
+#ifndef _LINUX_MEMCONTROL_H
+#define _LINUX_MEMCONTROL_H
+
+#ifdef CONFIG_CONTAINER_MEMCONTROL
+#ifndef kB
+#define kB 1024   /* One Kilo Byte */
+#endif
+
+#else /* CONFIG_CONTAINER_MEMCONTROL  */
+
+#endif /* CONFIG_CONTAINER_MEMCONTROL */
+#endif /* _LINUX_MEMCONTROL_H */
diff -puN init/Kconfig~memcontrol-setup init/Kconfig
--- linux-2.6.20/init/Kconfig~memcontrol-setup 2007-02-20 21:01:28.000000000 +0530
+++ linux-2.6.20-balbir/init/Kconfig 2007-02-20 21:01:28.000000000 +0530
@@ -306,6 +306,13 @@ config CONTAINER_NS
          for instance virtual servers and checkpoint/restart
          jobs.

+config CONTAINER_MEMCONTROL
+ 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~memcontrol-setup mm/Makefile
--- linux-2.6.20/mm/Makefile~memcontrol-setup 2007-02-20 21:01:28.000000000 +0530
+++ linux-2.6.20-balbir/mm/Makefile 2007-02-20 21:01:28.000000000 +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_MEMCONTROL) += memcontrol.o
diff -puN /dev/null mm/memcontrol.c
--- /dev/null 2007-02-02 22:51:23.000000000 +0530
+++ linux-2.6.20-balbir/mm/memcontrol.c 2007-02-24 19:39:24.000000000 +0530
@@ -0,0 +1,193 @@
+/*
+ * memcontrol.c - Memory Controller for containers
+ *
```

```
+ * 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.
+ *
+ * You should have received a copy of the GNU General Public License
+ * along with this program; if not, write to the Free Software
+ * Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111-1307, USA.
+ *
+ * © Copyright IBM Corporation, 2006-2007
+ *
+ * Author: Balbir Singh <balbir@in.ibm.com>
+ *
+ */
+
+#include <linux/init.h>
+#include <linux/parser.h>
+#include <linux/fs.h>
+#include <linux/container.h>
+#include <linux/memcontrol.h>
+
+#include <asm/uaccess.h>
+
+#define RES_USAGE_NO_LIMIT 0
+static const char version[] = "0.1";
+
+struct res_counter {
+ atomic_long_t usage; /* The current usage of the resource being */
+    /* counted        */
+ atomic_long_t limit; /* The limit on the resource     */
+};
+
+/*
+ * Each task belongs to a container, each container has a struct
+ * memcontrol associated with it
+ */
+struct memcontrol {
+ struct container_subsys_state css;
+ struct res_counter   counter;
+};
+
+static struct container_subsys memcontrol_subsys;
+
+static inline struct memcontrol *memcontrol_from_cont(struct container *cont)
+{
```

```
+ return container_of(container_subsys_state(cont, &memcontrol_subsys),
+     struct memcontrol, css);
+}
+
+static inline struct memcontrol *memcontrol_from_task(struct task_struct *p)
+{
+ return memcontrol_from_cont(task_container(p, &memcontrol_subsys));
+}
+
+static int memcontrol_create(struct container_subsys *ss,
+     struct container *cont)
+{
+ struct memcontrol *mem = kzalloc(sizeof(*mem), GFP_KERNEL);
+ if (!mem)
+  return -ENOMEM;
+
+ cont->subsys[memcontrol_subsys.subsys_id] = &mem->css;
+ atomic_long_set(&mem->counter.usage, 0);
+ atomic_long_set(&mem->counter.limit, 0);
+ return 0;
+}
+
+static void memcontrol_destroy(struct container_subsys *ss,
+     struct container *cont)
+{
+ kfree(memcontrol_from_cont(cont));
+}
+
+static ssize_t memcontrol_limit_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 memcontrol *mem = memcontrol_from_cont(cont);
+
+ BUG_ON(!mem);
+ buffer = kmalloc(nbytes + 1, GFP_KERNEL);
+ if (buffer == NULL)
+  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;
+
+ atomic_long_set(&mem->counter.limit, limit);
+ ret = nbytes;
+out_unlock:
+ container_manage_unlock();
+out_err:
+ kfree(buffer);
+ return ret;
+}
+
+static ssize_t memcontrol_limit_read(struct container *cont, struct cftype *cft,
+     struct file *file, char __user *userbuf,
+     size_t nbytes, loff_t *ppos)
+{
+ long limit;
+ char buf[64];  /* Move away from stack later */
+ char *s = buf;
+ struct memcontrol *mem = memcontrol_from_cont(cont);
+
+ limit = atomic_long_read(&mem->counter.limit);
+
+ s += sprintf(s, "limit: %ld pages (%ld kB)\n", limit,
+   (limit * PAGE_SIZE) / kB);
+ return simple_read_from_buffer(userbuf, nbytes, ppos, buf, s - buf);
+}
+
+static ssize_t memcontrol_usage_read(struct container *cont, struct cftype *cft,
+     struct file *file, char __user *userbuf,
+     size_t nbytes, loff_t *ppos)
+{
+ long usage;
+ char buf[64];  /* Move away from stack later */
+ char *s = buf;
+ struct memcontrol *mem = memcontrol_from_cont(cont);
+
```

```
+ usage = atomic_long_read(&mem->counter.usage);
+
+ s += sprintf(s, "usage: %ld pages (%ld kB)\n", usage,
+   (usage * PAGE_SIZE) / kB);
+ return simple_read_from_buffer(userbuf, nbytes, ppos, buf, s - buf);
+}
+
+static struct cftype memcontrol_usage = {
+ .name = "memcontrol_usage",
+ .read = memcontrol_usage_read,
+};
+
+static struct cftype memcontrol_limit = {
+ .name = "memcontrol_limit",
+ .write = memcontrol_limit_write,
+ .read = memcontrol_limit_read,
+};
+
+static int memcontrol_populate(struct container_subsys *ss,
+    struct container *cont)
+{
+ int rc;
+ if ((rc = container_add_file(cont, &memcontrol_usage)) < 0)
+  return rc;
+ if ((rc = container_add_file(cont, &memcontrol_limit)) < 0)
+  return rc;
+ return 0;
+}
+
+static struct container_subsys memcontrol_subsys = {
+ .name = "memcontrol",
+ .create = memcontrol_create,
+ .destroy = memcontrol_destroy,
+ .populate = memcontrol_populate,
+};
+
+int __init memcontrol_init(void)
+{
+ int id;
+
+ id = container_register_subsys(&memcontrol_subsys);
+ printk("Initializing memcontrol version %s, id %d\n", version, id);
+ return id < 0 ? id : 0;
+}
+
+module_init(memcontrol_init);
_
```

--
Warm Regards,
Balbir Singh

Subject: [RFC][PATCH][2/4] Add RSS accounting and control (
Posted by Balbir Singh on Sat, 24 Feb 2007 14:45:24 GMT
View Forum Message <> Reply to Message

Changelog

1. Be consistent, use the C style of returning 0 on success and negative
   values on failure
2. Change and document the locking used by the controller
   (I hope I got it right this time :-))
3. Remove memctlr_double_(un)lock routines
4. Comment the usage of MEMCONTROL_DONT_CHECK_LIMIT

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/memcontrol.h |   47 +++++++++++++++++
 include/linux/sched.h      |   11 +++
 kernel/fork.c              |   10 +++
 mm/memcontrol.c            |  130 +++++++++++++++++++++++++++++++++++++++++++++---
 mm/memory.c                |   34 ++++++++++-
 mm/rmap.c                  |    5 +
 mm/swapfile.c              |    2
 8 files changed, 234 insertions(+), 9 deletions(-)

diff -puN fs/exec.c~memcontrol-acct fs/exec.c
--- linux-2.6.20/fs/exec.c~memcontrol-acct 2007-02-24 19:39:29.000000000 +0530
+++ linux-2.6.20-balbir/fs/exec.c 2007-02-24 19:39:29.000000000 +0530
@@ -50,6 +50,7 @@
 #include <linux/tsacct_kern.h>
```

```
 #include <linux/cn_proc.h>
 #include <linux/audit.h>
+#include <linux/memcontrol.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 (memcontrol_update_rss(mm, 1, MEMCONTROL_CHECK_LIMIT))
+  goto out;
+
  flush_dcache_page(page);
  pte = get_locked_pte(mm, address, &ptl);
  if (!pte)
diff -puN include/linux/memcontrol.h~memcontrol-acct include/linux/memcontrol.h
--- linux-2.6.20/include/linux/memcontrol.h~memcontrol-acct 2007-02-24 19:39:29.000000000
+0530
+++ linux-2.6.20-balbir/include/linux/memcontrol.h 2007-02-24 19:39:29.000000000 +0530
@@ -22,12 +22,59 @@
 #ifndef _LINUX_MEMCONTROL_H
 #define _LINUX_MEMCONTROL_H

+/*
+ * MEMCONTROL_DONT_CHECK_LIMIT is useful for the following cases
+ * 1. During fork(), since pages are shared COW, we don't enforce limits
+ *    on fork
+ * 2. During zeromap_pte_range(), again we don't enforce the limit for
+ *    sharing ZERO_PAGE() in this case
+ * 3. When we actually reduce the RSS, add -1 to the rss
+ * It is generally useful when we do not want to enforce limits
+ */
+enum {
+ MEMCONTROL_CHECK_LIMIT = true,
+ MEMCONTROL_DONT_CHECK_LIMIT = false,
+};
+
 #ifdef CONFIG_CONTAINER_MEMCONTROL
+
 #ifndef kB
 #define kB 1024   /* One Kilo Byte */
 #endif

+struct res_counter {
+ atomic_long_t usage; /* The current usage of the resource being */
+    /* counted        */
+ atomic_long_t limit; /* The limit on the resource     */
```

```
+};
+
+extern int memcontrol_mm_init(struct mm_struct *mm);
+extern void memcontrol_mm_free(struct mm_struct *mm);
+extern void memcontrol_mm_assign_container(struct mm_struct *mm,
+       struct task_struct *p);
+extern int memcontrol_update_rss(struct mm_struct *mm, int count, bool check);
+
 #else /* CONFIG_CONTAINER_MEMCONTROL  */

+static inline int memcontrol_mm_init(struct mm_struct *mm)
+{
+ return 0;
+}
+
+static inline void memcontrol_mm_free(struct mm_struct *mm)
+{
+}
+
+static inline void memcontrol_mm_assign_container(struct mm_struct *mm,
+       struct task_struct *p)
+{
+}
+
+static inline int memcontrol_update_rss(struct mm_struct *mm, int count,
+     bool check)
+{
+ return 0;
+}
+
 #endif /* CONFIG_CONTAINER_MEMCONTROL */
 #endif /* _LINUX_MEMCONTROL_H */
diff -puN include/linux/sched.h~memcontrol-acct include/linux/sched.h
--- linux-2.6.20/include/linux/sched.h~memcontrol-acct 2007-02-24 19:39:29.000000000 +0530
+++ linux-2.6.20-balbir/include/linux/sched.h 2007-02-24 19:39:29.000000000 +0530
@@ -83,6 +83,7 @@ struct sched_param {
 #include <linux/timer.h>
 #include <linux/hrtimer.h>
 #include <linux/task_io_accounting.h>
+#include <linux/memcontrol.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_MEMCONTROL
```

```
+ /*
+  * 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_MEMCONTROL */
 };

 struct sighand_struct {
diff -puN kernel/fork.c~memcontrol-acct kernel/fork.c
--- linux-2.6.20/kernel/fork.c~memcontrol-acct 2007-02-24 19:39:29.000000000 +0530
+++ linux-2.6.20-balbir/kernel/fork.c 2007-02-24 19:39:29.000000000 +0530
@@ -50,6 +50,7 @@
 #include <linux/taskstats_kern.h>
 #include <linux/random.h>
 #include <linux/numtasks.h>
+#include <linux/memcontrol.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 (memcontrol_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)
+   memcontrol_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);
+ memcontrol_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;

+ memcontrol_mm_assign_container(mm, tsk);
  err = dup_mmap(mm, oldmm);
  if (err)
   goto free_pt;
diff -puN mm/memcontrol.c~memcontrol-acct mm/memcontrol.c
--- linux-2.6.20/mm/memcontrol.c~memcontrol-acct 2007-02-24 19:39:29.000000000 +0530
+++ linux-2.6.20-balbir/mm/memcontrol.c 2007-02-24 19:40:35.000000000 +0530
@@ -30,11 +30,20 @@
 #define RES_USAGE_NO_LIMIT 0
 static const char version[] = "0.1";

-struct res_counter {
- atomic_long_t usage; /* The current usage of the resource being */
-    /* counted        */
- atomic_long_t limit; /* The limit on the resource     */
-};
+/*
+ * Locking notes
+ *
+ * Each mm_struct belongs to a container, when the thread group leader
+ * moves from one container to another, the mm_struct's container is
+ * also migrated to the new container.
+ *
+ * We use a reader/writer lock for consistency. All changes to mm->container
+ * are protected by mm->container_lock. The lock prevents the use-after-free
+ * race condition that can occur. Without the lock, the mm->container
+ * pointer could change from under us.
+ *
+ * The counter uses atomic operations and does not require locking.
+ */

 /*
  * Each task belongs to a container, each container has a struct
@@ -58,11 +67,76 @@ static inline struct memcontrol *memcont
  return memcontrol_from_cont(task_container(p, &memcontrol_subsys));
 }
```

```
+int memcontrol_mm_init(struct mm_struct *mm)
+{
+ mm->counter = kmalloc(sizeof(struct res_counter), GFP_KERNEL);
+ if (mm->counter == NULL)
+  return -ENOMEM;
+ atomic_long_set(&mm->counter->usage, 0);
+ atomic_long_set(&mm->counter->limit, 0);
+ rwlock_init(&mm->container_lock);
+ return 0;
+}
+
+void memcontrol_mm_free(struct mm_struct *mm)
+{
+ kfree(mm->counter);
+}
+
+static inline void memcontrol_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 memcontrol_mm_assign_container(struct mm_struct *mm, struct task_struct *p)
+{
+ struct container *cont = task_container(p, &memcontrol_subsys);
+
+ BUG_ON(!cont);
+ memcontrol_mm_assign_container_direct(mm, cont);
+}
+
+/*
+ * 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 memcontrol_update_rss(struct mm_struct *mm, int count, bool check)
+{
+ int ret = 0;
+ struct container *cont;
+ long usage, limit;
+ struct memcontrol *mem;
+
+ read_lock(&mm->container_lock);
+ cont = mm->container;
+
+ if (cont == NULL)
```

```
+  goto out_unlock;
+
+ mem = memcontrol_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 = -ENOMEM; /* Above limit, fail */
+ else {
+  atomic_long_add(count, &mem->counter.usage);
+  atomic_long_add(count, &mm->counter->usage);
+ }
+
+out_unlock:
+ read_unlock(&mm->container_lock);
+ return ret;
+}
+
 static int memcontrol_create(struct container_subsys *ss,
     struct container *cont)
 {
  struct memcontrol *mem = kzalloc(sizeof(*mem), GFP_KERNEL);
- if (!mem)
+ if (mem == NULL)
   return -ENOMEM;

  cont->subsys[memcontrol_subsys.subsys_id] = &mem->css;
@@ -174,11 +248,55 @@ static int memcontrol_populate(struct co
  return 0;
 }

+int memcontrol_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
+ * baggage) The other possibility is move over last accounting information
+ * from mm_struct and charge the new container. We implement the latter.
+ */
```

```
+static void memcontrol_attach(struct container_subsys *ss,
+    struct container *cont,
+    struct container *old_cont,
+    struct task_struct *p)
+{
+ struct memcontrol *mem, *old_mem;
+ long usage;
+
+ if (cont == old_cont)
+  return;
+
+ mem = memcontrol_from_cont(cont);
+ old_mem = memcontrol_from_cont(old_cont);
+
+ memcontrol_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);
+}
+
 static struct container_subsys memcontrol_subsys = {
  .name = "memcontrol",
  .create = memcontrol_create,
  .destroy = memcontrol_destroy,
  .populate = memcontrol_populate,
+ .attach = memcontrol_attach,
+ .can_attach = memcontrol_can_attach,
 };

 int __init memcontrol_init(void)
diff -puN mm/memory.c~memcontrol-acct mm/memory.c
--- linux-2.6.20/mm/memory.c~memcontrol-acct 2007-02-24 19:39:29.000000000 +0530
+++ linux-2.6.20-balbir/mm/memory.c 2007-02-24 19:39:29.000000000 +0530
@@ -50,6 +50,7 @@
 #include <linux/delayacct.h>
 #include <linux/init.h>
 #include <linux/writeback.h>
+#include <linux/memcontrol.h>

 #include <asm/pgalloc.h>
 #include <asm/uaccess.h>
@@ -532,6 +533,8 @@ again:
  spin_unlock(src_ptl);
```

```
  pte_unmap_nested(src_pte - 1);
  add_mm_rss(dst_mm, rss[0], rss[1]);
+ memcontrol_update_rss(dst_mm, rss[0] + rss[1],
+    MEMCONTROL_DONT_CHECK_LIMIT);
  pte_unmap_unlock(dst_pte - 1, dst_ptl);
  cond_resched();
  if (addr != end)
@@ -1128,6 +1131,7 @@ static int zeromap_pte_range(struct mm_s
   page_cache_get(page);
   page_add_file_rmap(page);
   inc_mm_counter(mm, file_rss);
+  memcontrol_update_rss(mm, 1, MEMCONTROL_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 +1227,10 @@ static int insert_page(struct mm_struct
  if (PageAnon(page))
   goto out;
  retval = -ENOMEM;
+
+ if (memcontrol_update_rss(mm, 1, MEMCONTROL_CHECK_LIMIT))
+  goto out;
+
  flush_dcache_page(page);
  pte = get_locked_pte(mm, addr, &ptl);
  if (!pte)
@@ -1580,6 +1588,9 @@ gotten:
   cow_user_page(new_page, old_page, address, vma);
  }

+ if (memcontrol_update_rss(mm, 1, MEMCONTROL_CHECK_LIMIT))
+  goto oom;
+
  /*
   * Re-check the pte - we dropped the lock
   */
@@ -1612,7 +1623,9 @@ gotten:
   /* Free the old page.. */
   new_page = old_page;
   ret |= VM_FAULT_WRITE;
- }
+ } else
+  memcontrol_update_rss(mm, -1, MEMCONTROL_DONT_CHECK_LIMIT);
+
  if (new_page)
   page_cache_release(new_page);
  if (old_page)
@@ -2024,16 +2037,19 @@ static int do_swap_page(struct mm_struct
```

```
  mark_page_accessed(page);
  lock_page(page);

+ if (memcontrol_update_rss(mm, 1, MEMCONTROL_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 +2084,8 @@ unlock:
  pte_unmap_unlock(page_table, ptl);
 out:
  return ret;
+out_nomap_uncharge:
+ memcontrol_update_rss(mm, -1, MEMCONTROL_DONT_CHECK_LIMIT);
 out_nomap:
  pte_unmap_unlock(page_table, ptl);
  unlock_page(page);
@@ -2092,6 +2110,9 @@ static int do_anonymous_page(struct mm_s
   /* Allocate our own private page. */
   pte_unmap(page_table);

+  if (memcontrol_update_rss(mm, 1, MEMCONTROL_CHECK_LIMIT))
+   goto oom;
+
   if (unlikely(anon_vma_prepare(vma)))
    goto oom;
   page = alloc_zeroed_user_highpage(vma, address);
@@ -2108,6 +2129,8 @@ static int do_anonymous_page(struct mm_s
   lru_cache_add_active(page);
   page_add_new_anon_rmap(page, vma, address);
  } else {
+  memcontrol_update_rss(mm, 1, MEMCONTROL_DONT_CHECK_LIMIT);
+
   /* Map the ZERO_PAGE - vm_page_prot is readonly */
   page = ZERO_PAGE(address);
```

```
   page_cache_get(page);
@@ -2218,6 +2241,9 @@ retry:
    }
   }

+ if (memcontrol_update_rss(mm, 1, MEMCONTROL_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 +2253,7 @@ retry:
   if (mapping && unlikely(sequence != mapping->truncate_count)) {
    pte_unmap_unlock(page_table, ptl);
    page_cache_release(new_page);
+   memcontrol_update_rss(mm, -1, MEMCONTROL_DONT_CHECK_LIMIT);
    cond_resched();
    sequence = mapping->truncate_count;
    smp_rmb();
@@ -2265,6 +2292,7 @@ retry:
   } else {
    /* One of our sibling threads was faster, back out. */
    page_cache_release(new_page);
+   memcontrol_update_rss(mm, -1, MEMCONTROL_DONT_CHECK_LIMIT);
    goto unlock;
   }

diff -puN mm/rmap.c~memcontrol-acct mm/rmap.c
--- linux-2.6.20/mm/rmap.c~memcontrol-acct 2007-02-24 19:39:29.000000000 +0530
+++ linux-2.6.20-balbir/mm/rmap.c 2007-02-24 19:39:29.000000000 +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 MEMCONTROL_DONT_CHECK_LIMIT, it is ok to call
+  * this function under the pte lock (since we will not block in reclaim)
+  */
+ memcontrol_update_rss(vma->vm_mm, -1, MEMCONTROL_DONT_CHECK_LIMIT);
  }

  /*
diff -puN mm/swapfile.c~memcontrol-acct mm/swapfile.c
--- linux-2.6.20/mm/swapfile.c~memcontrol-acct 2007-02-24 19:39:29.000000000 +0530
+++ linux-2.6.20-balbir/mm/swapfile.c 2007-02-24 19:39:29.000000000 +0530
@@ -27,6 +27,7 @@
 #include <linux/mutex.h>
 #include <linux/capability.h>
```

```
 #include <linux/syscalls.h>
+#include <linux/memcontrol.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);
+ memcontrol_update_rss(vma->vm_mm, 1, MEMCONTROL_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 Sat, 24 Feb 2007 14:45:35 GMT
View Forum Message <> Reply to Message

Changelog

1. Move void *container to struct container (in scan_control and vmscan.c
   and rmap.c)
2. The last set of patches churned the LRU list, in this release, pages
   that can do not belong to the container are moved to a skipped_pages
   list. At the end of the isolation they are added back to the zone
   list using list_spice_tail (a new function added in list.h).
   The disadvantage of this approach is that pages moved to skipped_pages
   will not be available for general reclaim. General testing on UML
   and a powerpc box showed that the changes worked.

   Other alternatives tried
   ----
   a. Do not delete the page from lru list, but that quickly lead to
      a panic, since the page was on LRU and we released the lru_lock
      in page_in_container

TODO's

1. Try a per-container LRU list, but that would mean expanding the page
   struct or special tricks like overloading the LRU pointer. A per-container
   list would also make it more difficult to handle shared pages, as a
   page will belong to just one container at-a-time.

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/list.h       |   26 +++++++++
 include/linux/memcontrol.h |   12 ++++
 include/linux/rmap.h       |   20 ++++++-
 include/linux/swap.h       |    3 +
 mm/memcontrol.c            |  122 ++++++++++++++++++++++++++++++++++++++++++++----
 mm/migrate.c               |    2
 mm/rmap.c                  |  100 +++++++++++++++++++++++++++++++++++++-
 mm/vmscan.c                |  114 +++++++++++++++++++++++++++++++++++++++++++-----
 8 files changed, 370 insertions(+), 29 deletions(-)

diff -puN include/linux/memcontrol.h~memcontrol-reclaim-on-limit include/linux/memcontrol.h
---  linux-2.6.20/include/linux/memcontrol.h~memcontrol-reclaim-o n-limit 2007-02-24
19:40:56.000000000 +0530
+++ linux-2.6.20-balbir/include/linux/memcontrol.h 2007-02-24 19:50:34.000000000 +0530
@@ -37,6 +37,7 @@ enum {
 };

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

 #ifndef kB
 #define kB 1024   /* One Kilo Byte */
@@ -53,6 +54,9 @@ extern void memcontrol_mm_free(struct mm
 extern void memcontrol_mm_assign_container(struct mm_struct *mm,
       struct task_struct *p);
 extern int memcontrol_update_rss(struct mm_struct *mm, int count, bool check);
```

```
+extern int memcontrol_mm_overlimit(struct mm_struct *mm, void *sc_cont);
+extern wait_queue_head_t memcontrol_reclaim_wq;
+extern bool memcontrol_reclaim_in_progress;

 #else /* CONFIG_CONTAINER_MEMCONTROL  */

@@ -76,5 +80,13 @@ static inline int memcontrol_update_rss(
  return 0;
 }

+/*
+ * In the absence of memory control, we always free mappings.
+ */
+static inline int memcontrol_mm_overlimit(struct mm_struct *mm, void *sc_cont)
+{
+ return 1;
+}
+
 #endif /* CONFIG_CONTAINER_MEMCONTROL */
 #endif /* _LINUX_MEMCONTROL_H */
diff -puN include/linux/rmap.h~memcontrol-reclaim-on-limit include/linux/rmap.h
---  linux-2.6.20/include/linux/rmap.h~memcontrol-reclaim-on-limi t 2007-02-24
19:40:56.000000000 +0530
+++ linux-2.6.20-balbir/include/linux/rmap.h 2007-02-24 19:40:56.000000000 +0530
@@ -8,6 +8,7 @@
 #include <linux/slab.h>
 #include <linux/mm.h>
 #include <linux/spinlock.h>
+#include <linux/container.h>

 /*
  * The anon_vma heads a list of private "related" vmas, to scan if
@@ -90,7 +91,17 @@ 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, struct container *container);
+#ifdef CONFIG_CONTAINER_MEMCONTROL
+bool page_in_container(struct page *page, struct zone *zone,
+   struct container *container);
+#else
+static inline bool page_in_container(struct page *page, struct zone *zone,
+     struct container *container)
+{
+ return true;
+}
+#endif /* CONFIG_CONTAINER_MEMCONTROL */
```

```
 /*
  * Called from mm/filemap_xip.c to unmap empty zero page
@@ -118,7 +129,12 @@ 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
+static inline bool page_in_container(struct page *page, struct zone *zone,
+      struct container *container)
+{
+ return true;
+}

 static inline int page_mkclean(struct page *page)
 {
diff -puN include/linux/swap.h~memcontrol-reclaim-on-limit include/linux/swap.h
---  linux-2.6.20/include/linux/swap.h~memcontrol-reclaim-on-limi t 2007-02-24
19:40:56.000000000 +0530
+++ linux-2.6.20-balbir/include/linux/swap.h 2007-02-24 19:40:56.000000000 +0530
@@ -6,6 +6,7 @@
 #include <linux/mmzone.h>
 #include <linux/list.h>
 #include <linux/sched.h>
+#include <linux/container.h>

 #include <asm/atomic.h>
 #include <asm/page.h>
@@ -188,6 +189,8 @@ 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);
+extern unsigned long memcontrol_shrink_mapped_memory(unsigned long nr_pages,
+      struct container *container);
 extern int vm_swappiness;
 extern int remove_mapping(struct address_space *mapping, struct page *page);
 extern long vm_total_pages;
diff -puN mm/memcontrol.c~memcontrol-reclaim-on-limit mm/memcontrol.c
--- linux-2.6.20/mm/memcontrol.c~memcontrol-reclaim-on-limit 2007-02-24 19:40:56.000000000
+0530
+++ linux-2.6.20-balbir/mm/memcontrol.c 2007-02-24 19:40:56.000000000 +0530
@@ -24,6 +24,7 @@
 #include <linux/fs.h>
 #include <linux/container.h>
 #include <linux/memcontrol.h>
+#include <linux/swap.h>
```

```
 #include <asm/uaccess.h>

@@ -31,6 +32,12 @@
 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  */
+
+/*
  * Locking notes
  *
  * Each mm_struct belongs to a container, when the thread group leader
@@ -52,6 +59,9 @@ static const char version[] = "0.1";
 struct memcontrol {
  struct container_subsys_state css;
  struct res_counter   counter;
+ wait_queue_head_t  wq;
+ bool    reclaim_in_progress;
+ spinlock_t   lock;
 };

 static struct container_subsys memcontrol_subsys;
@@ -67,6 +77,41 @@ static inline struct memcontrol *memcont
  return memcontrol_from_cont(task_container(p, &memcontrol_subsys));
 }

+/*
+ * checks if the mm's container and scan control passed container match, if
+ * so, is the container over it's limit. Returns 1 to indicate that the
+ * pages from the mm_struct in question should be reclaimed.
+ */
+int memcontrol_mm_overlimit(struct mm_struct *mm, void *sc_cont)
+{
+ struct container *cont;
+ struct memcontrol *mem;
+ long usage, limit;
+ int ret = 1;
+
+ /*
+  * Regular reclaim, let it proceed as usual
+  */
+ if (!sc_cont)
+  goto out;
+
+ ret = 0;
```

```
+ read_lock(&mm->container_lock);
+ cont = mm->container;
+ if (cont != sc_cont)
+  goto out_unlock;
+
+ mem = memcontrol_from_cont(cont);
+ usage = atomic_long_read(&mem->counter.usage);
+ limit = atomic_long_read(&mem->counter.limit);
+ if (limit && (usage > limit))
+  ret = 1;
+out_unlock:
+ read_unlock(&mm->container_lock);
+out:
+ return ret;
+}
+
 int memcontrol_mm_init(struct mm_struct *mm)
 {
  mm->counter = kmalloc(sizeof(struct res_counter), GFP_KERNEL);
@@ -99,6 +144,46 @@ void memcontrol_mm_assign_container(stru
  memcontrol_mm_assign_container_direct(mm, cont);
 }

+static int memcontrol_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 = 0;
+ struct memcontrol *mem;
+
+ mem = memcontrol_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 +=
+    memcontrol_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 = -ENOMEM;
+ 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()).
@@ -120,12 +205,15 @@ int memcontrol_update_rss(struct mm_stru
  usage = atomic_long_read(&mem->counter.usage);
  limit = atomic_long_read(&mem->counter.limit);
  usage += count;
- if (check && limit && (usage > limit))
-  ret = -ENOMEM; /* Above limit, fail */
- else {
-  atomic_long_add(count, &mem->counter.usage);
-  atomic_long_add(count, &mm->counter->usage);
- }
+
+ if (check)
+  if (memcontrol_check_and_reclaim(cont, usage, limit)) {
+   ret = -ENOMEM;
+   goto out_unlock;
+  }
+
+ atomic_long_add(count, &mem->counter.usage);
+ atomic_long_add(count, &mm->counter->usage);

 out_unlock:
  read_unlock(&mm->container_lock);
@@ -142,6 +230,9 @@ static int memcontrol_create(struct cont
  cont->subsys[memcontrol_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 = false;
+ spin_lock_init(&mem->lock);
  return 0;
```

```
 }

@@ -157,8 +248,8 @@ static ssize_t memcontrol_limit_write(st
      size_t nbytes, loff_t *ppos)
 {
  char *buffer;
- int ret = 0;
- unsigned long limit;
+ int ret = nbytes;
+ unsigned long cur_limit, limit, usage;
  struct memcontrol *mem = memcontrol_from_cont(cont);

  BUG_ON(!mem);
@@ -186,7 +277,14 @@ static ssize_t memcontrol_limit_write(st
   goto out_unlock;

  atomic_long_set(&mem->counter.limit, limit);
- ret = nbytes;
+
+ usage = atomic_read(&mem->counter.usage);
+ cur_limit = atomic_read(&mem->counter.limit);
+ if (limit && (usage > limit))
+  if (memcontrol_check_and_reclaim(cont, usage, cur_limit)) {
+   ret = -EAGAIN; /* Try again, later */
+   goto out_unlock;
+  }
 out_unlock:
  container_manage_unlock();
 out_err:
@@ -276,6 +374,12 @@ static void memcontrol_attach(struct con
  if (cont == old_cont)
   return;

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

diff -puN mm/migrate.c~memcontrol-reclaim-on-limit mm/migrate.c
--- linux-2.6.20/mm/migrate.c~memcontrol-reclaim-on-limit 2007-02-24 19:40:56.000000000
+0530
+++ linux-2.6.20-balbir/mm/migrate.c 2007-02-24 19:40:56.000000000 +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~memcontrol-reclaim-on-limit mm/rmap.c
--- linux-2.6.20/mm/rmap.c~memcontrol-reclaim-on-limit 2007-02-24 19:40:56.000000000 +0530
+++ linux-2.6.20-balbir/mm/rmap.c 2007-02-24 19:40:56.000000000 +0530
@@ -792,7 +792,8 @@ 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,
+    struct container *container)
 {
  struct anon_vma *anon_vma;
  struct vm_area_struct *vma;
@@ -803,6 +804,13 @@ 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 (!memcontrol_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 +828,8 @@ 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,
+    struct container *container)
 {
  struct address_space *mapping = page->mapping;
  pgoff_t pgoff = page->index << (PAGE_CACHE_SHIFT - PAGE_SHIFT);
@@ -834,6 +843,12 @@ 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 (!memcontrol_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 +895,8 @@ static int try_to_unmap_file(struct page
       shared.vm_set.list) {
    if ((vma->vm_flags & VM_LOCKED) && !migration)
     continue;
+   if (!memcontrol_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 +936,92 @@ 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, struct container *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_MEMCONTROL
+bool anon_page_in_container(struct page *page, struct container *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 (memcontrol_mm_overlimit(vma->vm_mm, container)) {
+   ret = true;
+   break;
+  }
+
+ spin_unlock(&anon_vma->lock);
+ return ret;
+}
+
+bool file_page_in_container(struct page *page, struct container *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 (memcontrol_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 (memcontrol_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,
+   struct container *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~memcontrol-reclaim-on-limit mm/vmscan.c
--- linux-2.6.20/mm/vmscan.c~memcontrol-reclaim-on-limit 2007-02-24 19:40:56.000000000
+0530
+++ linux-2.6.20-balbir/mm/vmscan.c 2007-02-24 19:40:56.000000000 +0530
@@ -42,6 +42,7 @@
 #include <asm/div64.h>

 #include <linux/swapops.h>
+#include <linux/memcontrol.h>

 #include "internal.h"

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

  int all_unreclaimable;
+
+ struct container *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:
@@ -621,19 +625,43 @@ 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,
+   struct container *container, unsigned long max_scan)
 {
   unsigned long nr_taken = 0;
   struct page *page;
- unsigned long scan;
+ unsigned long scan, vscan;
+ LIST_HEAD(skipped_pages);

- 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);

    VM_BUG_ON(!PageLRU(page));

+   /*
+    * For containers, do not scan the page unless it
+    * belongs to the container we are reclaiming for
+    */
+   if (container) {
+    /*
+     * Move the page to skipped_pages list, since
+     * we give up the lru_lock in page_in_container()
+     * it is important to take page off LRU before
+     * the routine is called.
+     */
+    list_move(&page->lru, &skipped_pages);
+    if (!page_in_container(page, zone, container)) {
+     scan--;
+     /*
+      * Page continues to live in skipped pages
+      * It will be added back later to the LRU
+      */
+     continue;
+    }
+   }
    list_del(&page->lru);
    target = src;
    if (likely(get_page_unless_zero(page))) {
```

```
@@ -646,10 +674,17 @@ static unsigned long isolate_lru_pages(u
    target = dst;
    nr_taken++;
   } /* else it is being freed elsewhere */
-
   list_add(&page->lru, target);
+
  }

+ /*
+  * Add back the skipped pages in LRU order to avoid churning
+  * the LRU
+  */
+ if (container)
+  list_splice_tail(&skipped_pages, src);
+
  *scanned = scan;
  return nr_taken;
 }
@@ -678,7 +713,8 @@ 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 +859,8 @@ 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 +1398,7 @@ void wakeup_kswapd(struct zone *zone, in
  wake_up_interruptible(&pgdat->kswapd_wait);
 }

-#ifdef CONFIG_PM
+#if defined(CONFIG_PM) || defined(CONFIG_CONTAINER_MEMCONTROL)
 /*
  * 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 +1407,7 @@ 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 +1423,7 @@ 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 +1431,7 @@ 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 +1442,9 @@ 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 +1516,7 @@ 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 +1551,57 @@ out:
 }
```

```
 #endif

+#ifdef CONFIG_CONTAINER_MEMCONTROL
+/*
+ * Try to free `nr_pages' of memory, system-wide, and return the number of
+ * freed pages.
+ * Modelled after shrink_all_memory()
+ */
+unsigned long memcontrol_shrink_mapped_memory(unsigned long nr_pages,
+      struct container *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,
+  .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,
diff -puN include/linux/mm_types.h~memcontrol-reclaim-on-limit include/linux/mm_types.h
diff -puN include/linux/list.h~memcontrol-reclaim-on-limit include/linux/list.h
---  linux-2.6.20/include/linux/list.h~memcontrol-reclaim-on-limi t 2007-02-24 19:40:56.000000000
+0530
+++ linux-2.6.20-balbir/include/linux/list.h 2007-02-24 19:40:56.000000000 +0530
@@ -343,6 +343,32 @@ static inline void list_splice(struct li
   __list_splice(list, head);
 }

+static inline void __list_splice_tail(struct list_head *list,
+     struct list_head *head)
+{
+ struct list_head *first = list->next;
+ struct list_head *last = list->prev;
+ struct list_head *at = head->prev;
+
+ first->prev = at;
+ at->next = first;
+
+ last->next = head;
+ head->prev = last;
+}
+
+/**
+ * list_splice - join two lists, @list goes to the end (at head->prev)
+ * @list: the new list to add.
+ * @head: the place to add it in the first list.
+ */
+static inline void list_splice_tail(struct list_head *list,
+     struct list_head *head)
+{
+ if (!list_empty(list))
+  __list_splice_tail(list, head);
+}
+
 /**
  * list_splice_init - join two lists and reinitialise the emptied list.
  * @list: the new list to add.
_
```

--
Warm Regards,
Balbir Singh

Subject: [RFC][PATCH][4/4] RSS controller documentation (
Posted by Balbir Singh on Sat, 24 Feb 2007 14:45:46 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 2007-02-24 19:41:23.000000000 +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 memcontrol_usage file. The value
+is in units of pages.
+
+Control
+-------
+
+The memcontrol_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: Memcontrol patchset (was Re: [RFC][PATCH][0/4] Memory controller (RSS Control) ()
Posted by Balbir Singh on Sat, 24 Feb 2007 14:48:00 GMT
View Forum Message <> Reply to Message

Hi,

My script could not parse the (#2) and posted the patches as subject
followed by "(" instead

I apologize,
Balbir Singh