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
```