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