Subject: [RFC 0/4] memcg: background reclaim (v1)
Posted by KAMEZAWA Hiroyuki on Tue, 27 May 2008 05:01:16 GMT
View Forum Message <> Reply to Message

This is my current set of add-on patches for memory resource controller.
This works well but not well tested and not for usual people.
i.e..request for comments at early stage before being more complicated.

This set inculdes an implementation of background reclaim to memory resource
controller. I expect this helps I/O under memory resource controller very much.
(some good result with "dd")

pathces are based on 2.6.26-rc2-mm1 + remove_refcnt patch set (in mm queue)
So, I don't ask you "pleaset test" ;)
plz tell me if you don't like the concept or you have better idea.

[1/4] freeing all at force_empty.
[2/4] high-low watermark to resource counter.
[3/4] background reclaim for memcg.
[4/4] background reclaim for memcg, NUMA extension.

Consideration:
One problem of background reclaim is that it uses CPU. I think it's necessary
to make them more moderate. But what can I do against kthread rather than
nice() ?

Thanks,
-Kame

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

Subject: [RFC 1/4] memcg: drop pages at rmdir (v1)
Posted by KAMEZAWA Hiroyuki on Tue, 27 May 2008 05:04:30 GMT
View Forum Message <> Reply to Message

Now, when we remove memcg, we call force_empty().
This call drops all page_cgroup accounting in this mem_cgroup but doesn't
drop pages. So, some page caches can be remaind as "not accounted" memory
while they are alive. (because it's accounted only when add_to_page_cache())
If they are not used by other memcg, global LRU will drop them.

This patch tries to drop pages at removing memcg. Other memcg will
reload and re-account page caches. (but this will increase page-in
after rmdir().)

Consideration: should we recharge all pages to the parent at last ?
But it's not precise logic.

Changelog v1->v2
- renamed res_counter_empty().

Signed-off-by: KAMEZAWA Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>

---
include/linux/res_counter.h | 11 ++++++++++
mm/memcontrol.c | 19 ++++++++++++++++++++
2 files changed, 30 insertions(+)

Index: mm-2.6.26-rc2-mm1/mm/memcontrol.c
===
--- mm-2.6.26-rc2-mm1.orig/mm/memcontrol.c
+++ mm-2.6.26-rc2-mm1/mm/memcontrol.c
@@ -791,6 +791,20 @@ int mem_cgroup_shrink_usage(struct mm_st
return 0;
}

+
+static void mem_cgroup_drop_all_pages(struct mem_cgroup *mem)
+{
+ int progress;
+ while (!res_counter_empty(&mem->res)) {
+ progress = try_to_free_mem_cgroup_pages(mem,
+ GFP_HIGHUSER_MOVABLE);
+ if (!progress) /* we did as much as possible */
+ break;
+ cond_resched();
+ }
+ return;
+}
+
/*
* This routine traverse page_cgroup in given list and drop them all.
* *And* this routine doesn't reclaim page itself, just removes page_cgroup.
@@ -848,7 +862,12 @@ static int mem_cgroup_force_empty(struct
if (mem_cgroup_subsys.disabled)
return 0;

+ if (atomic_read(&mem->css.cgroup->count) > 0)
+ goto out;
+
css_get(&mem->css);
+ /* drop pages as much as possible */

```
+ mem_cgroup_drop_all_pages(mem);
  /*
   * page reclaim code (kswapd etc..) will move pages between
   * active_list <-> inactive_list while we don't take a lock.
Index: mm-2.6.26-rc2-mm1/include/linux/res_counter.h
===================================================================
--- mm-2.6.26-rc2-mm1.orig/include/linux/res_counter.h
+++ mm-2.6.26-rc2-mm1/include/linux/res_counter.h
@@ -153,4 +153,15 @@ static inline void res_counter_reset_fai
  cnt->failcnt = 0;
  spin_unlock_irqrestore(&cnt->lock, flags);
 }
+/* returns 0 if usage is 0. */
+static inline int res_counter_empty(struct res_counter *cnt)
+{
+ unsigned long flags;
+ int ret;
+
+ spin_lock_irqsave(&cnt->lock, flags);
+ ret = (cnt->usage == 0) ? 0 : 1;
+ spin_unlock_irqrestore(&cnt->lock, flags);
+ return ret;
+}
 #endif
```


Subject: [RFC 2/4] memcg: high-low watermark
Posted by KAMEZAWA Hiroyuki on Tue, 27 May 2008 05:06:39 GMT
View Forum Message <> Reply to Message

Add high/low watermarks to res_counter.
*This patch itself has no behavior changes to memory resource controller.

Changelog: very old one -> this one (v1)
 - watarmark_state is removed and all state check is done under lock.
 - changed res_counter_charge() interface. The only user is memory
   resource controller. Anyway, returning -ENOMEM here is a bit starnge.
 - Added watermark enable/disable flag for someone don't want watermarks.
 - Restarted against 2.6.25-mm1.
 - some subsystem which doesn't want high-low watermark can work withou it.

Signed-off-by: KAMEZAWA Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
From: YAMAMOTO Takashi <yamamoto@valinux.co.jp>

```
---
 include/linux/res_counter.h |   41 ++++++++++++++++++++---
 kernel/res_counter.c        |   66 ++++++++++++++++++++++++++++++++++++++----
 mm/memcontrol.c             |    2 -
 3 files changed, 99 insertions(+), 10 deletions(-)

Index: mm-2.6.26-rc2-mm1/include/linux/res_counter.h
===================================================================
--- mm-2.6.26-rc2-mm1.orig/include/linux/res_counter.h
+++ mm-2.6.26-rc2-mm1/include/linux/res_counter.h
@@ -16,6 +16,16 @@
 #include <linux/cgroup.h>

 /*
+ * status of resource coutner's usage.
+ */
+enum res_state {
+ RES_BELOW_LOW, /* usage < lwmark */
+ RES_BELOW_HIGH, /* lwmark < usage < hwmark */
+ RES_BELOW_LIMIT, /* hwmark < usage < limit. */
+ RES_OVER_LIMIT,  /* only used at chage. */
+};
+
+/*
  * The core object. the cgroup that wishes to account for some
  * resource may include this counter into its structures and use
  * the helpers described beyond
@@ -39,6 +49,12 @@ struct res_counter {
  */
  unsigned long long failcnt;
  /*
+  * watermarks. needs to keep lwmark <= hwmark <= limit.
+  */
+ unsigned long long hwmark;
+ unsigned long long lwmark;
+ int     use_watermark;
+ /*
  * the lock to protect all of the above.
  * the routines below consider this to be IRQ-safe
  */
@@ -76,13 +92,18 @@ enum {
  RES_MAX_USAGE,
  RES_LIMIT,
  RES_FAILCNT,
+ RES_HWMARK,
+ RES_LWMARK,
 };
```

```
 /*
  * helpers for accounting
+ * res_counter_init() ... initialize counter and disable watermarks.
+ * res_counter_init_wmark() ... initialize counter and enable watermarks.
  */

 void res_counter_init(struct res_counter *counter);
+void res_counter_init_wmark(struct res_counter *counter);

 /*
  * charge - try to consume more resource.
@@ -93,11 +114,21 @@ void res_counter_init(struct res_counter
  *
  * returns 0 on success and <0 if the counter->usage will exceed the
  * counter->limit _locked call expects the counter->lock to be taken
+ * return values:
+ * If watermark is disabled,
+ * RES_BELOW_LIMIT  --  usage is smaller than limt, success.
+ * RES_OVER_LIMIT   --  usage is bigger than limit, failed.
+ *
+ * If watermark is enabled,
+ * RES_BELOW_LOW    -- usage is smaller than low watermark, success
+ * RES_BELOW_HIGH   -- usage is smaller than high watermark, success.
+ * RES_BELOW_LIMIT  -- usage is smaller than limt, success.
+ * RES_OVER_LIMIT   -- usage is bigger than limit, failed.
  */

-int __must_check res_counter_charge_locked(struct res_counter *counter,
-  unsigned long val);
-int __must_check res_counter_charge(struct res_counter *counter,
+enum res_state __must_check
+res_counter_charge_locked(struct res_counter *counter, unsigned long val);
+enum res_state __must_check res_counter_charge(struct res_counter *counter,
    unsigned long val);

 /*
@@ -164,4 +195,7 @@ static inline int res_counter_empty(stru
  spin_unlock_irqrestore(&cnt->lock, flags);
  return ret;
 }
+
+enum res_state res_counter_state(struct res_counter *counter);
+
 #endif
Index: mm-2.6.26-rc2-mm1/kernel/res_counter.c
===================================================================
--- mm-2.6.26-rc2-mm1.orig/kernel/res_counter.c
```

```
+++ mm-2.6.26-rc2-mm1/kernel/res_counter.c
@@ -18,22 +18,40 @@ void res_counter_init(struct res_counter
 {
  spin_lock_init(&counter->lock);
  counter->limit = (unsigned long long)LLONG_MAX;
+ counter->use_watermark = 0;
 }

-int res_counter_charge_locked(struct res_counter *counter, unsigned long val)
+void res_counter_init_wmark(struct res_counter *counter)
+{
+ spin_lock_init(&counter->lock);
+ counter->limit = (unsigned long long)LLONG_MAX;
+ counter->hwmark = (unsigned long long)LLONG_MAX;
+ counter->lwmark = (unsigned long long)LLONG_MAX;
+ counter->use_watermark = 1;
+}
+
+enum res_state
+res_counter_charge_locked(struct res_counter *counter, unsigned long val)
 {
  if (counter->usage + val > counter->limit) {
   counter->failcnt++;
-  return -ENOMEM;
+  return RES_OVER_LIMIT;
  }

  counter->usage += val;
  if (counter->usage > counter->max_usage)
   counter->max_usage = counter->usage;
- return 0;
+ if (counter->use_watermark) {
+  if (counter->usage <= counter->lwmark)
+   return RES_BELOW_LOW;
+  if (counter->usage <= counter->hwmark)
+   return RES_BELOW_HIGH;
+ }
+ return RES_BELOW_LIMIT;
 }

-int res_counter_charge(struct res_counter *counter, unsigned long val)
+enum res_state
+res_counter_charge(struct res_counter *counter, unsigned long val)
 {
  int ret;
  unsigned long flags;
@@ -44,6 +62,23 @@ int res_counter_charge(struct res_counte
  return ret;
```

```
 }

+enum res_state res_counter_state(struct res_counter *counter)
+{
+ unsigned long flags;
+ enum res_state ret = RES_BELOW_LIMIT;
+
+ spin_lock_irqsave(&counter->lock, flags);
+ if (counter->use_watermark) {
+  if (counter->usage <= counter->lwmark)
+   ret = RES_BELOW_LOW;
+  else if (counter->usage <= counter->hwmark)
+   ret = RES_BELOW_HIGH;
+ }
+ spin_unlock_irqrestore(&counter->lock, flags);
+ return ret;
+}
+
+
 void res_counter_uncharge_locked(struct res_counter *counter, unsigned long val)
 {
  if (WARN_ON(counter->usage < val))
@@ -74,6 +109,10 @@ res_counter_member(struct res_counter *c
   return &counter->limit;
  case RES_FAILCNT:
   return &counter->failcnt;
+ case RES_HWMARK:
+  return &counter->hwmark;
+ case RES_LWMARK:
+  return &counter->lwmark;
  };

  BUG();
@@ -134,10 +173,27 @@ ssize_t res_counter_write(struct res_cou
    goto out_free;
  }
  spin_lock_irqsave(&counter->lock, flags);
+ switch (member) {
+  case RES_LIMIT:
+   if (counter->use_watermark && counter->hwmark > tmp)
+    goto unlock_free;
+   break;
+  case RES_HWMARK:
+   if (tmp < counter->lwmark  || tmp > counter->limit)
+    goto unlock_free;
+   break;
+  case RES_LWMARK:
+   if (tmp > counter->hwmark)
```

```
+    goto unlock_free;
+   break;
+  default:
+   break;
+ }
  val = res_counter_member(counter, member);
  *val = tmp;
- spin_unlock_irqrestore(&counter->lock, flags);
  ret = nbytes;
+unlock_free:
+ spin_unlock_irqrestore(&counter->lock, flags);
 out_free:
  kfree(buf);
 out:
Index: mm-2.6.26-rc2-mm1/mm/memcontrol.c
===================================================================
--- mm-2.6.26-rc2-mm1.orig/mm/memcontrol.c
+++ mm-2.6.26-rc2-mm1/mm/memcontrol.c
@@ -559,7 +559,7 @@ static int mem_cgroup_charge_common(stru
   css_get(&memcg->css);
  }

- while (res_counter_charge(&mem->res, PAGE_SIZE)) {
+ while (res_counter_charge(&mem->res, PAGE_SIZE) == RES_OVER_LIMIT) {
   if (!(gfp_mask & __GFP_WAIT))
    goto out;
```


Subject: [RFC 4/4] memcg: NUMA background reclaim
Posted by KAMEZAWA Hiroyuki on Tue, 27 May 2008 05:06:52 GMT
View Forum Message <> Reply to Message

One aspect of difference in reclaim logic between global lru and memcg is
 * global LRU triggers memory reclaim at memory shortage.
 * memcg LRU triggers memory reclaim at excess of usage.

Then, global LRU _know_ which node we should start reclaim from.
 * start from a node at memory shortage or
 * start from a node where memory allocation is waiting

WRT memcg, it's difficult to find where we should start because
there is no memory shortage and LRU is splitted.

(But per-zone-LRU is definitely necessary for scalability.)

This patch tries to deteremine a node for starting recalim by checking
ratio of inactive pages/active pages in a node. And trying to avoid starting
from a node with relatively small usage.
Better algorithm is welcome.

Singed-off-by: KAMEZAWA Hiruyuki <kamezawa.hiroyu@jp.fujitsu.com>

Index: mm-2.6.26-rc2-mm1/mm/memcontrol.c
```
===================================================================
--- mm-2.6.26-rc2-mm1.orig/mm/memcontrol.c
+++ mm-2.6.26-rc2-mm1/mm/memcontrol.c
@@ -578,7 +578,7 @@ retry:
  if (state == RES_OVER_LIMIT) {
   if (!(gfp_mask & __GFP_WAIT))
    goto out;
-  if (try_to_free_mem_cgroup_pages(mem, gfp_mask))
+  if (try_to_free_mem_cgroup_pages(mem, -1, gfp_mask))
    goto retry;
   /*
    * try_to_free_mem_cgroup_pages() might not give us a
@@ -801,7 +801,7 @@ int mem_cgroup_shrink_usage(struct mm_st
  rcu_read_unlock();

  do {
-  progress = try_to_free_mem_cgroup_pages(mem, gfp_mask);
+  progress = try_to_free_mem_cgroup_pages(mem, -1, gfp_mask);
  } while (!progress && --retry);

  if (!retry)
@@ -814,7 +814,7 @@ static void mem_cgroup_drop_all_pages(st
 {
  int progress;
  while (!res_counter_empty(&mem->res)) {
-  progress = try_to_free_mem_cgroup_pages(mem,
+  progress = try_to_free_mem_cgroup_pages(mem, -1,
       GFP_HIGHUSER_MOVABLE);
   if (!progress) /* we did as much as possible */
    break;
@@ -912,6 +912,62 @@ out:
 /*
  * background reclaim daemon.
  */
+
+#ifdef CONFIG_NUMA
+/*
+ * Because memory controller's memory reclaim doesn't come from memory shortage,
```

```
+ * we cannot know which node should be reclaimed in an easy way.
+ * This routine select a node with inactive pages to be a node for starting
+ * scanning.
+ */
+int __select_best_node(struct mem_cgroup *mem)
+{
+ int nid;
+ int best_node = -1;
+ unsigned long highest_inactive_ratio = 0;
+ unsigned long active, inactive, inactive_ratio, total, threshold, flags;
+ struct mem_cgroup_per_zone *mz;
+ int zid;
+
+ /*
+  * When a node's memory usage is smaller than
+    * total_usage/num_of_node * 75%, we don't select the node
+  */
+ total = mem->res.usage >> PAGE_SHIFT;
+ threshold = (total / num_node_state(N_HIGH_MEMORY)) * 3 / 4;
+
+ /*
+  * See nodemask.h, N_HIGH_MEMORY means that a node has memory
+  * can be used for user's memory.(i.e. not means HIGHMEM).
+  */
+ for_each_node_state(nid, N_HIGH_MEMORY) {
+  active = 0;
+  inactive = 0;
+
+  for (zid = 0; zid < MAX_NR_ZONES; zid++) {
+   mz = mem_cgroup_zoneinfo(mem, nid, zid);
+   spin_lock_irqsave(&mz->lru_lock, flags);
+   active += MEM_CGROUP_ZSTAT(mz, MEM_CGROUP_ZSTAT_ACTIVE);
+   inactive +=
+    MEM_CGROUP_ZSTAT(mz, MEM_CGROUP_ZSTAT_INACTIVE);
+   spin_unlock_irqrestore(&mz->lru_lock, flags);
+  }
+
+  if (active + inactive < threshold)
+   continue;
+  inactive_ratio = (inactive * 100) / (active + 1);
+  if (inactive_ratio > highest_inactive_ratio)
+   best_node = nid;
+ }
+ return best_node;
+}
+#else
+int __select_best_node(struct mem_cgroup *mem)
+{
```

```
+ return 0;
+}
+#endif
+
 static int mem_cgroup_reclaim_daemon(void *data)
 {
  DEFINE_WAIT(wait);
@@ -935,13 +991,9 @@ static int mem_cgroup_reclaim_daemon(voi
    continue;
   }
   finish_wait(&mem->daemon.waitq, &wait);
-  /*
-   * memory resource controller doesn't see NUMA memory usage
-   * balancing, becasue we cannot know what balancing is good.
-   * TODO: some annotation or heuristics to detect which node
-   * we should start reclaim from.
-   */
-  ret = try_to_free_mem_cgroup_pages(mem, GFP_HIGHUSER_MOVABLE);
+
+  ret = try_to_free_mem_cgroup_pages(mem,
+    __select_best_node(mem), GFP_HIGHUSER_MOVABLE);

   yield();
  }
Index: mm-2.6.26-rc2-mm1/mm/vmscan.c
===================================================================
--- mm-2.6.26-rc2-mm1.orig/mm/vmscan.c
+++ mm-2.6.26-rc2-mm1/mm/vmscan.c
@@ -1429,7 +1429,7 @@ unsigned long try_to_free_pages(struct z
 #ifdef CONFIG_CGROUP_MEM_RES_CTLR

 unsigned long try_to_free_mem_cgroup_pages(struct mem_cgroup *mem_cont,
-      gfp_t gfp_mask)
+        int nid, gfp_t gfp_mask)
 {
  struct scan_control sc = {
   .may_writepage = !laptop_mode,
@@ -1442,9 +1442,11 @@ unsigned long try_to_free_mem_cgroup_pag
  };
  struct zonelist *zonelist;

+ if (nid == -1)
+  nid = numa_node_id();
  sc.gfp_mask = (gfp_mask & GFP_RECLAIM_MASK) |
    (GFP_HIGHUSER_MOVABLE & ~GFP_RECLAIM_MASK);
- zonelist = NODE_DATA(numa_node_id())->node_zonelists;
+ zonelist = NODE_DATA(nid)->node_zonelists;
  return do_try_to_free_pages(zonelist, &sc);
```

```
 }
 #endif
Index: mm-2.6.26-rc2-mm1/include/linux/swap.h
===================================================================
--- mm-2.6.26-rc2-mm1.orig/include/linux/swap.h
+++ mm-2.6.26-rc2-mm1/include/linux/swap.h
@@ -184,7 +184,7 @@ extern void swap_setup(void);
 extern unsigned long try_to_free_pages(struct zonelist *zonelist, int order,
     gfp_t gfp_mask);
 extern unsigned long try_to_free_mem_cgroup_pages(struct mem_cgroup *mem,
-       gfp_t gfp_mask);
+       int nid, gfp_t gfp_mask);
 extern int __isolate_lru_page(struct page *page, int mode);
 extern unsigned long shrink_all_memory(unsigned long nr_pages);
 extern int vm_swappiness;
```

_____

Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

---

Subject: [RFC 3/4] memcg: background reclaim
Posted by KAMEZAWA Hiroyuki on Tue, 27 May 2008 05:08:46 GMT
View Forum Message <> Reply to Message

Do background reclaim based on high-low watermarks.

This feature helps smooth work of processes under memcg by reclaiming memory in the kernel thread. # of limitation failure at mem_cgroup_charge() will dramatically reduced. But this also means a CPU is continuously used for reclaiming memory.

This one is very simple. Anyway, we need to update this when we add new complexity to memcg.

Major logic:
- add high-low watermark support to memory resource controller.
- create a kernel thread for cgroup when hwmark is changed. (once)
- stop a kernel thread at rmdir().
- start background reclaim if res_counter is over high-watermark.
- stop background reclaim if res_coutner is below low-watermark.
- for reclaiiming, just calls try_to_free_mem_cgroup_pages().
- kthread for reclaim 's priority is nice(0). default is (-5).
  (weaker is better ?)
- kthread for reclaim calls yield() on each loop.

TODO:

  - add an interface to start/stop daemon ?
  - wise numa support
  - too small "low watermark" targe just consumes CPU. Should we warn ?
    and what is the best value for hwmark/lwmark in general....?

Chagelog: old one -> this (v1)
  - start a thread at write of hwmark.

Signed-off-by: KAMEZAWA Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>

---
 mm/memcontrol.c | 147 ++-------
 1 file changed, 130 insertions(+), 17 deletions(-)

Index: mm-2.6.26-rc2-mm1/mm/memcontrol.c
===
--- mm-2.6.26-rc2-mm1.orig/mm/memcontrol.c
+++ mm-2.6.26-rc2-mm1/mm/memcontrol.c
@@ -32,7 +32,8 @@
 #include <linux/fs.h>
 #include <linux/seq_file.h>
 #include <linux/vmalloc.h>
-
+#include <linux/freezer.h>
+#include <linux/kthread.h>
 #include <asm/uaccess.h>

 struct cgroup_subsys mem_cgroup_subsys __read_mostly;
@@ -119,10 +120,6 @@ struct mem_cgroup_lru_info {
  * statistics based on the statistics developed by Rik Van Riel for clock-pro,
  * to help the administrator determine what knobs to tune.
  *
- * TODO: Add a water mark for the memory controller. Reclaim will begin when
- * we hit the water mark. May be even add a low water mark, such that
- * no reclaim occurs from a cgroup at it's low water mark, this is
- * a feature that will be implemented much later in the future.
  */
 struct mem_cgroup {
  struct cgroup_subsys_state css;
@@ -131,6 +128,13 @@ struct mem_cgroup {
   */
  struct res_counter res;
  /*
+ * background reclaim.
+ */
+ struct {
+ wait_queue_head_t waitq;

```
+  struct task_struct *thread;
+ } daemon;
+ /*
   * Per cgroup active and inactive list, similar to the
   * per zone LRU lists.
   */
@@ -143,6 +147,7 @@ struct mem_cgroup {
  struct mem_cgroup_stat stat;
 };
 static struct mem_cgroup init_mem_cgroup;
+static DEFINE_MUTEX(memcont_daemon_lock);

 /*
  * We use the lower bit of the page->page_cgroup pointer as a bit spin
@@ -374,6 +379,15 @@ void mem_cgroup_move_lists(struct page *
  unlock_page_cgroup(page);
 }

+static void mem_cgroup_schedule_reclaim(struct mem_cgroup *mem)
+{
+ if (!mem->daemon.thread)
+  return;
+ if (!waitqueue_active(&mem->daemon.waitq))
+  return;
+ wake_up_interruptible(&mem->daemon.waitq);
+}
+
 /*
  * Calculate mapped_ratio under memory controller. This will be used in
  * vmscan.c for deteremining we have to reclaim mapped pages.
@@ -532,6 +546,7 @@ static int mem_cgroup_charge_common(stru
  unsigned long flags;
  unsigned long nr_retries = MEM_CGROUP_RECLAIM_RETRIES;
  struct mem_cgroup_per_zone *mz;
+ enum res_state state;

  if (mem_cgroup_subsys.disabled)
   return 0;
@@ -558,23 +573,23 @@ static int mem_cgroup_charge_common(stru
   mem = memcg;
   css_get(&memcg->css);
  }
-
- while (res_counter_charge(&mem->res, PAGE_SIZE) == RES_OVER_LIMIT) {
+retry:
+ state = res_counter_charge(&mem->res, PAGE_SIZE);
+ if (state == RES_OVER_LIMIT) {
   if (!(gfp_mask & __GFP_WAIT))
```

```
    goto out;
-
   if (try_to_free_mem_cgroup_pages(mem, gfp_mask))
-   continue;
-
+   goto retry;
   /*
-   * try_to_free_mem_cgroup_pages() might not give us a full
-   * picture of reclaim. Some pages are reclaimed and might be
-   * moved to swap cache or just unmapped from the cgroup.
-   * Check the limit again to see if the reclaim reduced the
-   * current usage of the cgroup before giving up
+    * try_to_free_mem_cgroup_pages() might not give us a
+    * full picture of reclaim. Some pages are reclaimed
+    * and might be moved to swap cache or just unmapped
+    * from the cgroup. Check the limit again to see if
+    * the reclaim reduced the current usage of the cgroup
+    * before giving up
     */
   if (res_counter_check_under_limit(&mem->res))
-   continue;
+   goto retry;

   if (!nr_retries--) {
    mem_cgroup_out_of_memory(mem, gfp_mask);
@@ -609,6 +624,9 @@ static int mem_cgroup_charge_common(stru
  spin_unlock_irqrestore(&mz->lru_lock, flags);

  unlock_page_cgroup(page);
+
+ if (state > RES_BELOW_HIGH)
+  mem_cgroup_schedule_reclaim(mem);
 done:
  return 0;
 out:
@@ -891,6 +909,74 @@ out:
  css_put(&mem->css);
  return ret;
 }
+/*
+ * background reclaim daemon.
+ */
+static int mem_cgroup_reclaim_daemon(void *data)
+{
+ DEFINE_WAIT(wait);
+ struct mem_cgroup *mem = data;
+ int ret;
+
```

```
+ css_get(&mem->css);
+ current->flags |= PF_SWAPWRITE;
+ /* we don't want to use cpu too much. */
+  set_user_nice(current, 0);
+ set_freezable();
+
+ while (!kthread_should_stop()) {
+  prepare_to_wait(&mem->daemon.waitq, &wait, TASK_INTERRUPTIBLE);
+  if (res_counter_state(&mem->res) == RES_BELOW_LOW) {
+   if (!kthread_should_stop()) {
+    schedule();
+    try_to_freeze();
+   }
+   finish_wait(&mem->daemon.waitq, &wait);
+   continue;
+  }
+  finish_wait(&mem->daemon.waitq, &wait);
+  /*
+   * memory resource controller doesn't see NUMA memory usage
+   * balancing, becasue we cannot know what balancing is good.
+   * TODO: some annotation or heuristics to detect which node
+   * we should start reclaim from.
+   */
+  ret = try_to_free_mem_cgroup_pages(mem, GFP_HIGHUSER_MOVABLE);
+
+  yield();
+ }
+ css_put(&mem->css);
+ return 0;
+}
+
+static int mem_cgroup_start_daemon(struct mem_cgroup *mem)
+{
+ int ret = 0;
+ struct task_struct *thr;
+
+ mutex_lock(&memcont_daemon_lock);
+ if (!mem->daemon.thread) {
+  thr = kthread_run(mem_cgroup_reclaim_daemon, mem, "memcontd");
+  if (IS_ERR(thr))
+   ret = PTR_ERR(thr);
+  else
+   mem->daemon.thread = thr;
+ }
+ mutex_unlock(&memcont_daemon_lock);
+ return ret;
+}
+
```

```
+static void mem_cgroup_stop_daemon(struct mem_cgroup *mem)
+{
+ mutex_lock(&memcont_daemon_lock);
+ if (mem->daemon.thread) {
+  kthread_stop(mem->daemon.thread);
+  mem->daemon.thread = NULL;
+ }
+ mutex_unlock(&memcont_daemon_lock);
+ return;
+}
+

 static int mem_cgroup_write_strategy(char *buf, unsigned long long *tmp)
 {
@@ -915,6 +1001,19 @@ static ssize_t mem_cgroup_write(struct c
     struct file *file, const char __user *userbuf,
     size_t nbytes, loff_t *ppos)
 {
+ int ret;
+ /*
+  * start daemon can fail. But we should start daemon always
+  * when changes to HWMARK is succeeded. So, we start daemon before
+  * changes to HWMARK. We don't stop this daemon even if
+  * res_counter_write fails. To do that, we need ugly codes and
+  * it's not so big problem.
+  */
+ if (cft->private == RES_HWMARK) {
+  ret = mem_cgroup_start_daemon(mem_cgroup_from_cont(cont));
+  if (ret)
+   return ret;
+ }
  return res_counter_write(&mem_cgroup_from_cont(cont)->res,
     cft->private, userbuf, nbytes, ppos,
     mem_cgroup_write_strategy);
@@ -1004,6 +1103,18 @@ static struct cftype mem_cgroup_files[]
   .read_u64 = mem_cgroup_read,
  },
  {
+  .name = "high_wmark_in_bytes",
+  .private = RES_HWMARK,
+  .write = mem_cgroup_write,
+  .read_u64 = mem_cgroup_read,
+ },
+ {
+  .name = "low_wmark_in_bytes",
+  .private = RES_LWMARK,
+  .write = mem_cgroup_write,
+  .read_u64 = mem_cgroup_read,
```

```
+ },
+ {
   .name = "force_empty",
   .trigger = mem_force_empty_write,
  },
@@ -1087,7 +1198,8 @@ mem_cgroup_create(struct cgroup_subsys *
    return ERR_PTR(-ENOMEM);
  }

- res_counter_init(&mem->res);
+ res_counter_init_wmark(&mem->res);
+ init_waitqueue_head(&mem->daemon.waitq);

  for_each_node_state(node, N_POSSIBLE)
   if (alloc_mem_cgroup_per_zone_info(mem, node))
@@ -1106,6 +1218,7 @@ static void mem_cgroup_pre_destroy(struc
      struct cgroup *cont)
 {
  struct mem_cgroup *mem = mem_cgroup_from_cont(cont);
+ mem_cgroup_stop_daemon(mem);
  mem_cgroup_force_empty(mem);
 }
```

Subject: Re: [RFC 2/4] memcg: high-low watermark
Posted by yamamoto on Tue, 27 May 2008 05:30:27 GMT
View Forum Message <> Reply to Message

```
> +enum res_state res_counter_state(struct res_counter *counter)
> +{
> + unsigned long flags;
> + enum res_state ret = RES_BELOW_LIMIT;
> +
> + spin_lock_irqsave(&counter->lock, flags);
> + if (counter->use_watermark) {
> +  if (counter->usage <= counter->lwmark)
> +   ret = RES_BELOW_LOW;
> +  else if (counter->usage <= counter->hwmark)
> +   ret = RES_BELOW_HIGH;
> + }
> + spin_unlock_irqrestore(&counter->lock, flags);
> + return ret;
```

> +}

can't it be RES_OVER_LIMIT?
eg. when you lower the limit.

YAMAMOTO Takashi

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

---

Subject: Re: [RFC 2/4] memcg: high-low watermark
Posted by KAMEZAWA Hiroyuki on Tue, 27 May 2008 07:14:30 GMT
View Forum Message <> Reply to Message

On Tue, 27 May 2008 14:30:27 +0900 (JST)
yamamoto@valinux.co.jp (YAMAMOTO Takashi) wrote:

> > +enum res_state res_counter_state(struct res_counter *counter)
> > +{
> > + unsigned long flags;
> > + enum res_state ret = RES_BELOW_LIMIT;
> > +
> > + spin_lock_irqsave(&counter->lock, flags);
> > + if (counter->use_watermark) {
> > + if (counter->usage <= counter->lwmark)
> > + ret = RES_BELOW_LOW;
> > + else if (counter->usage <= counter->hwmark)
> > + ret = RES_BELOW_HIGH;
> > + }
> > + spin_unlock_irqrestore(&counter->lock, flags);
> > + return ret;
> > +}
>
> can't it be RES_OVER_LIMIT?
> eg. when you lower the limit.
Ah, ok. I missed it. I'll add checks.

Thank you !

-Kame

_____

Subject: Re: [RFC 2/4] memcg: high-low watermark
Posted by Li Zefan on Tue, 27 May 2008 07:51:40 GMT
View Forum Message <> Reply to Message

KAMEZAWA Hiroyuki wrote:
> Add high/low watermarks to res_counter.
> *This patch itself has no behavior changes to memory resource controller.
>
> Changelog: very old one -> this one (v1)
> - watarmark_state is removed and all state check is done under lock.
> - changed res_counter_charge() interface. The only user is memory
>   resource controller. Anyway, returning -ENOMEM here is a bit starnge.
> - Added watermark enable/disable flag for someone don't want watermarks.
> - Restarted against 2.6.25-mm1.
> - some subsystem which doesn't want high-low watermark can work withou it.
>
> Signed-off-by: KAMEZAWA Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
> From: YAMAMOTO Takashi <yamamoto@valinux.co.jp>
>
> ---
> include/linux/res_counter.h | 41 ++++++++++++++++++++++++++---
> kernel/res_counter.c | 66 ++----
> mm/memcontrol.c | 2 -
> 3 files changed, 99 insertions(+), 10 deletions(-)
>
> Index: mm-2.6.26-rc2-mm1/include/linux/res_counter.h
> ===
> --- mm-2.6.26-rc2-mm1.orig/include/linux/res_counter.h
> +++ mm-2.6.26-rc2-mm1/include/linux/res_counter.h
> @@ -16,6 +16,16 @@
> #include <linux/cgroup.h>
>
> /*
> + * status of resource coutner's usage.
> + */
> +enum res_state {
> + RES_BELOW_LOW, /* usage < lwmark */

It seems it's 'usage <= lwmark'

> + RES_BELOW_HIGH, /* lwmark < usage < hwmark */

and 'lwmark < usage <= hwmark'

> + RES_BELOW_LIMIT, /* hwmark < usage < limit. */

and 'hwmark < usage <= limit'

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

---

Subject: Re: [RFC 2/4] memcg: high-low watermark
Posted by KAMEZAWA Hiroyuki on Tue, 27 May 2008 09:42:15 GMT
View Forum Message <> Reply to Message

On Tue, 27 May 2008 15:51:40 +0800
Li Zefan <lizf@cn.fujitsu.com> wrote:

> KAMEZAWA Hiroyuki wrote:
> > Add high/low watermarks to res_counter.
> > *This patch itself has no behavior changes to memory resource controller.
> >
> > Changelog: very old one -> this one (v1)
> > - watarmark_state is removed and all state check is done under lock.
> > - changed res_counter_charge() interface. The only user is memory
> >   resource controller. Anyway, returning -ENOMEM here is a bit starnge.
> > - Added watermark enable/disable flag for someone don't want watermarks.
> > - Restarted against 2.6.25-mm1.
> > - some subsystem which doesn't want high-low watermark can work withou it.
> >
> > Signed-off-by: KAMEZAWA Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
> > From: YAMAMOTO Takashi <yamamoto@valinux.co.jp>
> >
> > ---
> > include/linux/res_counter.h | 41 ++++++++++++++++++++++++---
> > kernel/res_counter.c | 66 ++++++++++++++++++++++++++++++++++++++----
> > mm/memcontrol.c | 2 -
> > 3 files changed, 99 insertions(+), 10 deletions(-)
> >
> > Index: mm-2.6.26-rc2-mm1/include/linux/res_counter.h
> > ===
> > --- mm-2.6.26-rc2-mm1.orig/include/linux/res_counter.h
> > +++ mm-2.6.26-rc2-mm1/include/linux/res_counter.h
> > @@ -16,6 +16,16 @@
> > #include <linux/cgroup.h>
> >
> > /*
> > + * status of resource coutner's usage.
> > + */

> > +enum res_state {
> > + RES_BELOW_LOW, /* usage < lwmark */
>
> It seems it's 'usage <= lwmark'
>
> > + RES_BELOW_HIGH, /* lwmark < usage < hwmark */
>
> and 'lwmark < usage <= hwmark'
>
> > + RES_BELOW_LIMIT, /* hwmark < usage < limit. */
>
> and 'hwmark < usage <= limit'
>

Thank you. I'll fix.

-Kame

Subject: Re: [RFC 1/4] memcg: drop pages at rmdir (v1)
Posted by Balbir Singh on Tue, 27 May 2008 16:11:26 GMT
View Forum Message <> Reply to Message

KAMEZAWA Hiroyuki wrote:
> Now, when we remove memcg, we call force_empty().
> This call drops all page_cgroup accounting in this mem_cgroup but doesn't
> drop pages. So, some page caches can be remaind as "not accounted" memory
> while they are alive. (because it's accounted only when add_to_page_cache())
> If they are not used by other memcg, global LRU will drop them.
>
> This patch tries to drop pages at removing memcg. Other memcg will
> reload and re-account page caches. (but this will increase page-in
> after rmdir().)
>

The approach seems fair, but I am not sure about the overhead of flushing out
cached pages. Might well be worth it.

> Consideration: should we recharge all pages to the parent at last ?
> But it's not precise logic.
>

We should look into this - I should send out the multi-hierarchy patches soon.
We should discuss this after that.

> Changelog v1->v2
> - renamed res_counter_empty().
>
> Signed-off-by: KAMEZAWA Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
>
> ---
> include/linux/res_counter.h | 11 ++++++++++
> mm/memcontrol.c | 19 +++++++++++++++++++
> 2 files changed, 30 insertions(+)
>
> Index: mm-2.6.26-rc2-mm1/mm/memcontrol.c
> ===
> --- mm-2.6.26-rc2-mm1.orig/mm/memcontrol.c
> +++ mm-2.6.26-rc2-mm1/mm/memcontrol.c
> @@ -791,6 +791,20 @@ int mem_cgroup_shrink_usage(struct mm_st
>   return 0;
> }
>
> +
> +static void mem_cgroup_drop_all_pages(struct mem_cgroup *mem)
> +{
> + int progress;
> + while (!res_counter_empty(&mem->res)) {
> + progress = try_to_free_mem_cgroup_pages(mem,
> +   GFP_HIGHUSER_MOVABLE);
> + if (!progress) /* we did as much as possible */
> +   break;
> + cond_resched();
> + }
> + return;
> +}
> +
> /*
>   * This routine traverse page_cgroup in given list and drop them all.
>   * *And* this routine doesn't reclaim page itself, just removes page_cgroup.
> @@ -848,7 +862,12 @@ static int mem_cgroup_force_empty(struct
>   if (mem_cgroup_subsys.disabled)
>   return 0;
>
> + if (atomic_read(&mem->css.cgroup->count) > 0)
> + goto out;
> +
>   css_get(&mem->css);
> + /* drop pages as much as possible */

> + mem_cgroup_drop_all_pages(mem);
>   /*
>   * page reclaim code (kswapd etc..) will move pages between
>   * active_list <-> inactive_list while we don't take a lock.
> Index: mm-2.6.26-rc2-mm1/include/linux/res_counter.h
> ===
> --- mm-2.6.26-rc2-mm1.orig/include/linux/res_counter.h
> +++ mm-2.6.26-rc2-mm1/include/linux/res_counter.h
> @@ -153,4 +153,15 @@ static inline void res_counter_reset_fai
>   cnt->failcnt = 0;
>   spin_unlock_irqrestore(&cnt->lock, flags);
> }
> +/* returns 0 if usage is 0. */
> +static inline int res_counter_empty(struct res_counter *cnt)
> +{
> + unsigned long flags;
> + int ret;
> +
> + spin_lock_irqsave(&cnt->lock, flags);
> + ret = (cnt->usage == 0) ? 0 : 1;
> + spin_unlock_irqrestore(&cnt->lock, flags);
> + return ret;
> +}
> #endif
>

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

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

---

Subject: Re: [RFC 2/4] memcg: high-low watermark
Posted by Balbir Singh on Tue, 27 May 2008 16:26:17 GMT
View Forum Message <> Reply to Message

KAMEZAWA Hiroyuki wrote:
> Add high/low watermarks to res_counter.
> *This patch itself has no behavior changes to memory resource controller.
>
> Changelog: very old one -> this one (v1)
> - watarmark_state is removed and all state check is done under lock.

> - changed res_counter_charge() interface. The only user is memory
>   resource controller. Anyway, returning -ENOMEM here is a bit starnge.
> - Added watermark enable/disable flag for someone don't want watermarks.
> - Restarted against 2.6.25-mm1.
> - some subsystem which doesn't want high-low watermark can work withou it.
>
> Signed-off-by: KAMEZAWA Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
> From: YAMAMOTO Takashi <yamamoto@valinux.co.jp>
>

The From: line should be the first line IIRC.

> ---
> include/linux/res_counter.h | 41 ++++++++++++++++++++++++---
> kernel/res_counter.c | 66 ++----
> mm/memcontrol.c | 2 -
> 3 files changed, 99 insertions(+), 10 deletions(-)
>
> Index: mm-2.6.26-rc2-mm1/include/linux/res_counter.h
> ===
> --- mm-2.6.26-rc2-mm1.orig/include/linux/res_counter.h
> +++ mm-2.6.26-rc2-mm1/include/linux/res_counter.h
> @@ -16,6 +16,16 @@
> #include <linux/cgroup.h>
>
> /*
> + * status of resource coutner's usage.
> + */
> +enum res_state {
> + RES_BELOW_LOW, /* usage < lwmark */
> + RES_BELOW_HIGH, /* lwmark < usage < hwmark */
> + RES_BELOW_LIMIT, /* hwmark < usage < limit. */
> + RES_OVER_LIMIT, /* only used at chage. */
> +};
> +
> +/*
>   * The core object. the cgroup that wishes to account for some
>   * resource may include this counter into its structures and use
>   * the helpers described beyond
> @@ -39,6 +49,12 @@ struct res_counter {
>   */
>   unsigned long long failcnt;
>   /*
> + * watermarks. needs to keep lwmark <= hwmark <= limit.
> + */
> + unsigned long long hwmark;
> + unsigned long long lwmark;
> + int   use_watermark;

Is it routine to comment this way? I prefer not to have spaces in the type and
the member, makes it easier for my eyes.

> + /*
> * the lock to protect all of the above.
> * the routines below consider this to be IRQ-safe
> */
> @@ -76,13 +92,18 @@ enum {
> RES_MAX_USAGE,
> RES_LIMIT,
> RES_FAILCNT,
> + RES_HWMARK,
> + RES_LWMARK,
> };
>
> /*
> * helpers for accounting
> + * res_counter_init() ... initialize counter and disable watermarks.
> + * res_counter_init_wmark() ... initialize counter and enable watermarks.
> */
>
> void res_counter_init(struct res_counter *counter);
> +void res_counter_init_wmark(struct res_counter *counter);
>
> /*
> * charge - try to consume more resource.
> @@ -93,11 +114,21 @@ void res_counter_init(struct res_counter
> *
> * returns 0 on success and <0 if the counter->usage will exceed the
> * counter->limit _locked call expects the counter->lock to be taken
> + * return values:
> + * If watermark is disabled,
> + * RES_BELOW_LIMIT -- usage is smaller than limt, success.

     ^^^^ typo

> + * RES_OVER_LIMIT -- usage is bigger than limit, failed.
> + *
> + * If watermark is enabled,
> + * RES_BELOW_LOW -- usage is smaller than low watermark, success
> + * RES_BELOW_HIGH -- usage is smaller than high watermark, success.
> + * RES_BELOW_LIMIT -- usage is smaller than limt, success.
> + * RES_OVER_LIMIT -- usage is bigger than limit, failed.
> */
>
> -int __must_check res_counter_charge_locked(struct res_counter *counter,
> - unsigned long val);

> -int __must_check res_counter_charge(struct res_counter *counter,
> +enum res_state __must_check
> +res_counter_charge_locked(struct res_counter *counter, unsigned long val);
> +enum res_state __must_check res_counter_charge(struct res_counter *counter,
> unsigned long val);
>
> /*
> @@ -164,4 +195,7 @@ static inline int res_counter_empty(stru
> spin_unlock_irqrestore(&cnt->lock, flags);
> return ret;
> }
> +
> +enum res_state res_counter_state(struct res_counter *counter);
> +
> #endif
> Index: mm-2.6.26-rc2-mm1/kernel/res_counter.c
> ===
> --- mm-2.6.26-rc2-mm1.orig/kernel/res_counter.c
> +++ mm-2.6.26-rc2-mm1/kernel/res_counter.c
> @@ -18,22 +18,40 @@ void res_counter_init(struct res_counter
> {
> spin_lock_init(&counter->lock);
> counter->limit = (unsigned long long)LLONG_MAX;
> + counter->use_watermark = 0;
> }
>
> -int res_counter_charge_locked(struct res_counter *counter, unsigned long val)
> +void res_counter_init_wmark(struct res_counter *counter)
> +{
> + spin_lock_init(&counter->lock);
> + counter->limit = (unsigned long long)LLONG_MAX;
> + counter->hwmark = (unsigned long long)LLONG_MAX;
> + counter->lwmark = (unsigned long long)LLONG_MAX;
> + counter->use_watermark = 1;
> +}
> +
> +enum res_state
> +res_counter_charge_locked(struct res_counter *counter, unsigned long val)
> {
> if (counter->usage + val > counter->limit) {
> counter->failcnt++;
> - return -ENOMEM;
> + return RES_OVER_LIMIT;
> }
>
> counter->usage += val;
> if (counter->usage > counter->max_usage)
> counter->max_usage = counter->usage;

> - return 0;
> + if (counter->use_watermark) {
> + if (counter->usage <= counter->lwmark)
> +   return RES_BELOW_LOW;
> + if (counter->usage <= counter->hwmark)
> +   return RES_BELOW_HIGH;
> + }
> + return RES_BELOW_LIMIT;
> }
>
> -int res_counter_charge(struct res_counter *counter, unsigned long val)
> +enum res_state
> +res_counter_charge(struct res_counter *counter, unsigned long val)
> {
>   int ret;
>   unsigned long flags;
> @@ -44,6 +62,23 @@ int res_counter_charge(struct res_counte
>   return ret;
> }
>
> +enum res_state res_counter_state(struct res_counter *counter)
> +{
> + unsigned long flags;
> + enum res_state ret = RES_BELOW_LIMIT;
> +
> + spin_lock_irqsave(&counter->lock, flags);
> + if (counter->use_watermark) {
> + if (counter->usage <= counter->lwmark)
> +   ret = RES_BELOW_LOW;
> + else if (counter->usage <= counter->hwmark)
> +   ret = RES_BELOW_HIGH;
> + }
> + spin_unlock_irqrestore(&counter->lock, flags);
> + return ret;
> +}
> +

When do we return RES_OVER_LIMIT? Are we missing that here?

> +
> void res_counter_uncharge_locked(struct res_counter *counter, unsigned long val)
> {
>   if (WARN_ON(counter->usage < val))
> @@ -74,6 +109,10 @@ res_counter_member(struct res_counter *c
>   return &counter->limit;
>   case RES_FAILCNT:
>   return &counter->failcnt;
> + case RES_HWMARK:

> + return &counter->hwmark;
> + case RES_LWMARK:
> + return &counter->lwmark;
>   };
>
>   BUG();
> @@ -134,10 +173,27 @@ ssize_t res_counter_write(struct res_cou
>   goto out_free;
>   }
>   spin_lock_irqsave(&counter->lock, flags);
> + switch (member) {
> + case RES_LIMIT:
> +   if (counter->use_watermark && counter->hwmark > tmp)
> +   goto unlock_free;

We need to document such API changes in the Documentation/controllers/memory.txt file.

> +   break;
> + case RES_HWMARK:
> +   if (tmp < counter->lwmark || tmp > counter->limit)
> +   goto unlock_free;
> +   break;
> + case RES_LWMARK:
> +   if (tmp > counter->hwmark)
> +   goto unlock_free;
> +   break;
> + default:
> +   break;
> + }
>   val = res_counter_member(counter, member);
>   *val = tmp;
> - spin_unlock_irqrestore(&counter->lock, flags);
>   ret = nbytes;
> +unlock_free:
> + spin_unlock_irqrestore(&counter->lock, flags);
> out_free:
>   kfree(buf);
> out:
> Index: mm-2.6.26-rc2-mm1/mm/memcontrol.c
> ===
> --- mm-2.6.26-rc2-mm1.orig/mm/memcontrol.c
> +++ mm-2.6.26-rc2-mm1/mm/memcontrol.c
> @@ -559,7 +559,7 @@ static int mem_cgroup_charge_common(stru
>   css_get(&memcg->css);
>   }
>
> - while (res_counter_charge(&mem->res, PAGE_SIZE)) {

> + while (res_counter_charge(&mem->res, PAGE_SIZE) == RES_OVER_LIMIT) {
> if (!(gfp_mask & __GFP_WAIT))
> goto out;
>

Otherwise looks good so far. Need to look at the background reclaim code.

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

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

---

Subject: Re: [RFC 3/4] memcg: background reclaim
Posted by Balbir Singh on Tue, 27 May 2008 17:08:09 GMT
View Forum Message <> Reply to Message

KAMEZAWA Hiroyuki wrote:
> Do background reclaim based on high-low watermarks.
>
> This feature helps smooth work of processes under memcg by reclaiming memory
> in the kernel thread. # of limitation failure at mem_cgroup_charge() will
> dramatically reduced. But this also means a CPU is continuously used for
> reclaiming memory.
>
> This one is very simple. Anyway, we need to update this when we add new
> complexity to memcg.
>
> Major logic:
> - add high-low watermark support to memory resource controller.
> - create a kernel thread for cgroup when hwmark is changed. (once)
> - stop a kernel thread at rmdir().
> - start background reclaim if res_counter is over high-watermark.
> - stop background reclaim if res_coutner is below low-watermark.
> - for reclaiiming, just calls try_to_free_mem_cgroup_pages().
> - kthread for reclaim 's priority is nice(0). default is (-5).
> (weaker is better ?)
> - kthread for reclaim calls yield() on each loop.
>
> TODO:
> - add an interface to start/stop daemon ?
> - wise numa support
> - too small "low watermark" targe just consumes CPU. Should we warn ?

>   and what is the best value for hwmark/lwmark in general....?
>
> Chagelog: old one -> this (v1)
>   - start a thread at write of hwmark.
>
>
> Signed-off-by: KAMEZAWA Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
>
> ---
> mm/memcontrol.c | 147 ++-------
> 1 file changed, 130 insertions(+), 17 deletions(-)
>
> Index: mm-2.6.26-rc2-mm1/mm/memcontrol.c
> ===
> --- mm-2.6.26-rc2-mm1.orig/mm/memcontrol.c
> +++ mm-2.6.26-rc2-mm1/mm/memcontrol.c
> @@ -32,7 +32,8 @@
> #include <linux/fs.h>
> #include <linux/seq_file.h>
> #include <linux/vmalloc.h>
> -
> +#include <linux/freezer.h>
> +#include <linux/kthread.h>
> #include <asm/uaccess.h>
>
> struct cgroup_subsys mem_cgroup_subsys __read_mostly;
> @@ -119,10 +120,6 @@ struct mem_cgroup_lru_info {
>   * statistics based on the statistics developed by Rik Van Riel for clock-pro,
>   * to help the administrator determine what knobs to tune.
>   *
> - * TODO: Add a water mark for the memory controller. Reclaim will begin when
> - * we hit the water mark. May be even add a low water mark, such that
> - * no reclaim occurs from a cgroup at it's low water mark, this is
> - * a feature that will be implemented much later in the future.
>   */
> struct mem_cgroup {
>   struct cgroup_subsys_state css;
> @@ -131,6 +128,13 @@ struct mem_cgroup {
>   */
>   struct res_counter res;
>   /*
> + * background reclaim.
> + */
> + struct {
> + wait_queue_head_t waitq;
> + struct task_struct *thread;
> + } daemon;

Comments on each of the members would be nice.

> + /*
> * Per cgroup active and inactive list, similar to the
> * per zone LRU lists.
> */
> @@ -143,6 +147,7 @@ struct mem_cgroup {
> struct mem_cgroup_stat stat;
> };
> static struct mem_cgroup init_mem_cgroup;
> +static DEFINE_MUTEX(memcont_daemon_lock);
>
> /*
> * We use the lower bit of the page->page_cgroup pointer as a bit spin
> @@ -374,6 +379,15 @@ void mem_cgroup_move_lists(struct page *
> unlock_page_cgroup(page);
> }
>
> +static void mem_cgroup_schedule_reclaim(struct mem_cgroup *mem)
> +{
> + if (!mem->daemon.thread)
> + return;

I suspect we are using threads. Aren't workqueues better than threads?

> + if (!waitqueue_active(&mem->daemon.waitq))
> + return;
> + wake_up_interruptible(&mem->daemon.waitq);
> +}
> +
> /*
> * Calculate mapped_ratio under memory controller. This will be used in
> * vmscan.c for deteremining we have to reclaim mapped pages.
> @@ -532,6 +546,7 @@ static int mem_cgroup_charge_common(stru
> unsigned long flags;
> unsigned long nr_retries = MEM_CGROUP_RECLAIM_RETRIES;
> struct mem_cgroup_per_zone *mz;
> + enum res_state state;
>
> if (mem_cgroup_subsys.disabled)
> return 0;
> @@ -558,23 +573,23 @@ static int mem_cgroup_charge_common(stru
> mem = memcg;
> css_get(&memcg->css);
> }
> -
> - while (res_counter_charge(&mem->res, PAGE_SIZE) == RES_OVER_LIMIT) {

> +retry:
> + state = res_counter_charge(&mem->res, PAGE_SIZE);
> + if (state == RES_OVER_LIMIT) {
>   if (!(gfp_mask & __GFP_WAIT))
>   goto out;
> -
>   if (try_to_free_mem_cgroup_pages(mem, gfp_mask))
> -   continue;
> -
> +   goto retry;
>   /*
> -   * try_to_free_mem_cgroup_pages() might not give us a full
> -   * picture of reclaim. Some pages are reclaimed and might be
> -   * moved to swap cache or just unmapped from the cgroup.
> -   * Check the limit again to see if the reclaim reduced the
> -   * current usage of the cgroup before giving up
> +   * try_to_free_mem_cgroup_pages() might not give us a
> +   * full picture of reclaim. Some pages are reclaimed
> +   * and might be moved to swap cache or just unmapped
> +   * from the cgroup. Check the limit again to see if
> +   * the reclaim reduced the current usage of the cgroup
> +   * before giving up
>   */
>   if (res_counter_check_under_limit(&mem->res))
> -   continue;
> +   goto retry;
>
>   if (!nr_retries--) {
>   mem_cgroup_out_of_memory(mem, gfp_mask);
> @@ -609,6 +624,9 @@ static int mem_cgroup_charge_common(stru
>   spin_unlock_irqrestore(&mz->lru_lock, flags);
>
>   unlock_page_cgroup(page);
> +
> + if (state > RES_BELOW_HIGH)
> + mem_cgroup_schedule_reclaim(mem);

I really don't like state > RES_BELOW_HIGH sort of checks. Could we please
abstract them into functions, like

if (mem_res_above_high_watermark(state))
 ....

> done:
>   return 0;
> out:
> @@ -891,6 +909,74 @@ out:
>   css_put(&mem->css);

> return ret;
> }
> +/*
> + * background reclaim daemon.
> + */
> +static int mem_cgroup_reclaim_daemon(void *data)
> +{
> + DEFINE_WAIT(wait);
> + struct mem_cgroup *mem = data;
> + int ret;
> +
> + css_get(&mem->css);
> + current->flags |= PF_SWAPWRITE;
> + /* we don't want to use cpu too much. */
> + set_user_nice(current, 0);

Shouldn't this (0) be a #define, what if we would like to degrade our nice value
even further later?

> + set_freezable();
> +
> + while (!kthread_should_stop()) {
> + prepare_to_wait(&mem->daemon.waitq, &wait, TASK_INTERRUPTIBLE);
> + if (res_counter_state(&mem->res) == RES_BELOW_LOW) {
> +   if (!kthread_should_stop()) {
> +   schedule();
> +   try_to_freeze();

I am afraid, I fail to understand the code above.

> +   }
> +   finish_wait(&mem->daemon.waitq, &wait);
> +   continue;
> + }
> + finish_wait(&mem->daemon.waitq, &wait);
> + /*
> +   * memory resource controller doesn't see NUMA memory usage
> +   * balancing, becasue we cannot know what balancing is good.

        ^^^^^^^^ typo

> +   * TODO: some annotation or heuristics to detect which node
> +   * we should start reclaim from.
> +   */
> + ret = try_to_free_mem_cgroup_pages(mem, GFP_HIGHUSER_MOVABLE);
> +
> + yield();

Why do we yeild here? Shouldn't our waitqueue handle the waiting?

```
> + }
> + css_put(&mem->css);
> + return 0;
> +}
> +
> +static int mem_cgroup_start_daemon(struct mem_cgroup *mem)
> +{
> + int ret = 0;
> + struct task_struct *thr;
> +
> + mutex_lock(&memcont_daemon_lock);
> + if (!mem->daemon.thread) {
> +  thr = kthread_run(mem_cgroup_reclaim_daemon, mem, "memcontd");
> +  if (IS_ERR(thr))
> +   ret = PTR_ERR(thr);
> +  else
> +   mem->daemon.thread = thr;
> + }
> + mutex_unlock(&memcont_daemon_lock);
> + return ret;
> +}
> +
> +static void mem_cgroup_stop_daemon(struct mem_cgroup *mem)
> +{
> + mutex_lock(&memcont_daemon_lock);
> + if (mem->daemon.thread) {
> +  kthread_stop(mem->daemon.thread);
> +  mem->daemon.thread = NULL;
> + }
> + mutex_unlock(&memcont_daemon_lock);
> + return;
> +}
> +
>
>  static int mem_cgroup_write_strategy(char *buf, unsigned long long *tmp)
>  {
> @@ -915,6 +1001,19 @@ static ssize_t mem_cgroup_write(struct c
>       struct file *file, const char __user *userbuf,
>       size_t nbytes, loff_t *ppos)
>  {
> + int ret;
> + /*
> +  * start daemon can fail. But we should start daemon always
> +  * when changes to HWMARK is succeeded. So, we start daemon before
> +  * changes to HWMARK. We don't stop this daemon even if
> +  * res_counter_write fails. To do that, we need ugly codes and
```

```
> +  * it's not so big problem.
> +  */
> + if (cft->private == RES_HWMARK) {
> +  ret = mem_cgroup_start_daemon(mem_cgroup_from_cont(cont));
> +  if (ret)
> +   return ret;
> + }
>   return res_counter_write(&mem_cgroup_from_cont(cont)->res,
>       cft->private, userbuf, nbytes, ppos,
>       mem_cgroup_write_strategy);
> @@ -1004,6 +1103,18 @@ static struct cftype mem_cgroup_files[]
>     .read_u64 = mem_cgroup_read,
>    },
>    {
> +  .name = "high_wmark_in_bytes",
> +  .private = RES_HWMARK,
> +  .write = mem_cgroup_write,
> +  .read_u64 = mem_cgroup_read,
> + },
> + {
> +  .name = "low_wmark_in_bytes",
> +  .private = RES_LWMARK,
> +  .write = mem_cgroup_write,
> +  .read_u64 = mem_cgroup_read,
> + },
> + {
>     .name = "force_empty",
>     .trigger = mem_force_empty_write,
>    },
> @@ -1087,7 +1198,8 @@ mem_cgroup_create(struct cgroup_subsys *
>      return ERR_PTR(-ENOMEM);
>    }
>
> - res_counter_init(&mem->res);
> + res_counter_init_wmark(&mem->res);
> + init_waitqueue_head(&mem->daemon.waitq);
>
>    for_each_node_state(node, N_POSSIBLE)
>     if (alloc_mem_cgroup_per_zone_info(mem, node))
> @@ -1106,6 +1218,7 @@ static void mem_cgroup_pre_destroy(struc
>        struct cgroup *cont)
>  {
>    struct mem_cgroup *mem = mem_cgroup_from_cont(cont);
> + mem_cgroup_stop_daemon(mem);
>    mem_cgroup_force_empty(mem);
>  }
>
```

I failed to see code for the low watermark. Shouldn't we stop the thread once we
go below the low watermark?

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

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

---

Subject: Re: [RFC 4/4] memcg: NUMA background reclaim
Posted by Balbir Singh on Tue, 27 May 2008 17:26:43 GMT
View Forum Message <> Reply to Message

KAMEZAWA Hiroyuki wrote:
> One aspect of difference in reclaim logic between global lru and memcg is
> * global LRU triggers memory reclaim at memory shortage.
> * memcg LRU triggers memory reclaim at excess of usage.
>
> Then, global LRU _know_ which node we should start reclaim from.
> * start from a node at memory shortage or
> * start from a node where memory allocation is waiting
>
> WRT memcg, it's difficult to find where we should start because
> there is no memory shortage and LRU is splitted.
> (But per-zone-LRU is definitely necessary for scalability.)
>
> This patch tries to deteremine a node for starting recalim by checking
> ratio of inactive pages/active pages in a node. And trying to avoid starting
> from a node with relatively small usage.
> Better algorithm is welcome.
>
> Singed-off-by: KAMEZAWA Hiruyuki <kamezawa.hiroyu@jp.fujitsu.com>
>
> Index: mm-2.6.26-rc2-mm1/mm/memcontrol.c
> ===
> --- mm-2.6.26-rc2-mm1.orig/mm/memcontrol.c
> +++ mm-2.6.26-rc2-mm1/mm/memcontrol.c
> @@ -578,7 +578,7 @@ retry:
> if (state == RES_OVER_LIMIT) {
> if (!(gfp_mask & __GFP_WAIT))
> goto out;
> - if (try_to_free_mem_cgroup_pages(mem, gfp_mask))

```
> +  if (try_to_free_mem_cgroup_pages(mem, -1, gfp_mask))
>     goto retry;
>    /*
>     * try_to_free_mem_cgroup_pages() might not give us a
> @@ -801,7 +801,7 @@ int mem_cgroup_shrink_usage(struct mm_st
>   rcu_read_unlock();
>
>   do {
> -  progress = try_to_free_mem_cgroup_pages(mem, gfp_mask);
> +  progress = try_to_free_mem_cgroup_pages(mem, -1, gfp_mask);
>   } while (!progress && --retry);
>
>   if (!retry)
> @@ -814,7 +814,7 @@ static void mem_cgroup_drop_all_pages(st
>  {
>   int progress;
>   while (!res_counter_empty(&mem->res)) {
> -  progress = try_to_free_mem_cgroup_pages(mem,
> +  progress = try_to_free_mem_cgroup_pages(mem, -1,
>       GFP_HIGHUSER_MOVABLE);
>    if (!progress) /* we did as much as possible */
>     break;
> @@ -912,6 +912,62 @@ out:
>  /*
>   * background reclaim daemon.
>   */
> +
> +#ifdef CONFIG_NUMA
> +/*
> + * Because memory controller's memory reclaim doesn't come from memory shortage,
> + * we cannot know which node should be reclaimed in an easy way.
> + * This routine select a node with inactive pages to be a node for starting
> + * scanning.
> + */
> +int __select_best_node(struct mem_cgroup *mem)
> +{
> + int nid;
> + int best_node = -1;
> + unsigned long highest_inactive_ratio = 0;
> + unsigned long active, inactive, inactive_ratio, total, threshold, flags;
> + struct mem_cgroup_per_zone *mz;
> + int zid;
> +
> + /*
> +  * When a node's memory usage is smaller than
> +  * total_usage/num_of_node * 75%, we don't select the node
> +  */
> + total = mem->res.usage >> PAGE_SHIFT;
```

```
> + threshold = (total / num_node_state(N_HIGH_MEMORY)) * 3 / 4;
> +
> + /*
> +  * See nodemask.h, N_HIGH_MEMORY means that a node has memory
> +  * can be used for user's memory.(i.e. not means HIGHMEM).
> +  */
> + for_each_node_state(nid, N_HIGH_MEMORY) {
> +  active = 0;
> +  inactive = 0;
> +
> +  for (zid = 0; zid < MAX_NR_ZONES; zid++) {
> +   mz = mem_cgroup_zoneinfo(mem, nid, zid);
> +   spin_lock_irqsave(&mz->lru_lock, flags);
> +   active += MEM_CGROUP_ZSTAT(mz, MEM_CGROUP_ZSTAT_ACTIVE);
> +   inactive +=
> +    MEM_CGROUP_ZSTAT(mz, MEM_CGROUP_ZSTAT_INACTIVE);
> +   spin_unlock_irqrestore(&mz->lru_lock, flags);
> +  }
> +
> +  if (active + inactive < threshold)
> +   continue;
> +  inactive_ratio = (inactive * 100) / (active + 1);
> +  if (inactive_ratio > highest_inactive_ratio)
> +   best_node = nid;
```

Shouldn't we update highest_inactive_ration here?

```
> + }
> + return best_node;
> +}
> +#else
> +int __select_best_node(struct mem_cgroup *mem)
> +{
> + return 0;
> +}
> +#endif
> +
>  static int mem_cgroup_reclaim_daemon(void *data)
>  {
>   DEFINE_WAIT(wait);
> @@ -935,13 +991,9 @@ static int mem_cgroup_reclaim_daemon(voi
>     continue;
>    }
>    finish_wait(&mem->daemon.waitq, &wait);
> -  /*
> -   * memory resource controller doesn't see NUMA memory usage
> -   * balancing, becasue we cannot know what balancing is good.
> -   * TODO: some annotation or heuristics to detect which node
```

```
> -   * we should start reclaim from.
> -   */
> -  ret = try_to_free_mem_cgroup_pages(mem, GFP_HIGHUSER_MOVABLE);
> +
> +  ret = try_to_free_mem_cgroup_pages(mem,
> +     __select_best_node(mem), GFP_HIGHUSER_MOVABLE);
>
>     yield();
>    }
> Index: mm-2.6.26-rc2-mm1/mm/vmscan.c
> ===================================================================
> --- mm-2.6.26-rc2-mm1.orig/mm/vmscan.c
> +++ mm-2.6.26-rc2-mm1/mm/vmscan.c
> @@ -1429,7 +1429,7 @@ unsigned long try_to_free_pages(struct z
>   #ifdef CONFIG_CGROUP_MEM_RES_CTLR
>
>   unsigned long try_to_free_mem_cgroup_pages(struct mem_cgroup *mem_cont,
> -       gfp_t gfp_mask)
> +       int nid, gfp_t gfp_mask)
>   {
>    struct scan_control sc = {
>     .may_writepage = !laptop_mode,
> @@ -1442,9 +1442,11 @@ unsigned long try_to_free_mem_cgroup_pag
>    };
>    struct zonelist *zonelist;
>
> + if (nid == -1)
> +  nid = numa_node_id();
>    sc.gfp_mask = (gfp_mask & GFP_RECLAIM_MASK) |
>      (GFP_HIGHUSER_MOVABLE & ~GFP_RECLAIM_MASK);
> - zonelist = NODE_DATA(numa_node_id())->node_zonelists;
> + zonelist = NODE_DATA(nid)->node_zonelists;
>    return do_try_to_free_pages(zonelist, &sc);
>   }
>   #endif
> Index: mm-2.6.26-rc2-mm1/include/linux/swap.h
> ===================================================================
> --- mm-2.6.26-rc2-mm1.orig/include/linux/swap.h
> +++ mm-2.6.26-rc2-mm1/include/linux/swap.h
> @@ -184,7 +184,7 @@ extern void swap_setup(void);
>   extern unsigned long try_to_free_pages(struct zonelist *zonelist, int order,
>        gfp_t gfp_mask);
>   extern unsigned long try_to_free_mem_cgroup_pages(struct mem_cgroup *mem,
> -       gfp_t gfp_mask);
> +       int nid, gfp_t gfp_mask);
>   extern int __isolate_lru_page(struct page *page, int mode);
>   extern unsigned long shrink_all_memory(unsigned long nr_pages);
>   extern int vm_swappiness;
```

>

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

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

---

Subject: Re: [RFC 1/4] memcg: drop pages at rmdir (v1)
Posted by KAMEZAWA Hiroyuki on Wed, 28 May 2008 00:12:37 GMT
View Forum Message <> Reply to Message

On Tue, 27 May 2008 21:41:26 +0530
Balbir Singh <balbir@linux.vnet.ibm.com> wrote:

> KAMEZAWA Hiroyuki wrote:
> > Now, when we remove memcg, we call force_empty().
> > This call drops all page_cgroup accounting in this mem_cgroup but doesn't
> > drop pages. So, some page caches can be remaind as "not accounted" memory
> > while they are alive. (because it's accounted only when add_to_page_cache())
> > If they are not used by other memcg, global LRU will drop them.
> >
> > This patch tries to drop pages at removing memcg. Other memcg will
> > reload and re-account page caches. (but this will increase page-in
> > after rmdir().)
> >
>
> The approach seems fair, but I am not sure about the overhead of flushing out
> cached pages. Might well be worth it.
>
> > Consideration: should we recharge all pages to the parent at last ?
> > But it's not precise logic.
> >
>
> We should look into this - I should send out the multi-hierarchy patches soon.
Yes. I'll write my version (if I can). please pick it up if you like it.

> We should discuss this after that.
>
ok. I'd like to forget this patch for a while.

Thanks,

-Kame

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

Subject: Re: [RFC 2/4] memcg: high-low watermark
Posted by KAMEZAWA Hiroyuki on Wed, 28 May 2008 00:13:30 GMT
View Forum Message <> Reply to Message

On Tue, 27 May 2008 21:56:17 +0530
Balbir Singh <balbir@linux.vnet.ibm.com> wrote:

> KAMEZAWA Hiroyuki wrote:
> > Add high/low watermarks to res_counter.
> > *This patch itself has no behavior changes to memory resource controller.
> >
> > Changelog: very old one -> this one (v1)
> > - watarmark_state is removed and all state check is done under lock.
> > - changed res_counter_charge() interface. The only user is memory
> >   resource controller. Anyway, returning -ENOMEM here is a bit starnge.
> > - Added watermark enable/disable flag for someone don't want watermarks.
> > - Restarted against 2.6.25-mm1.
> > - some subsystem which doesn't want high-low watermark can work withou it.
> >
> > Signed-off-by: KAMEZAWA Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
> > From: YAMAMOTO Takashi <yamamoto@valinux.co.jp>
> >
>
> The From: line should be the first line IIRC.
>
ok.

>
> >   /*
> > + * watermarks. needs to keep lwmark <= hwmark <= limit.
> > + */
> > + unsigned long long hwmark;
> > + unsigned long long lwmark;
> > + int   use_watermark;
>

> Is it routine to comment this way? I prefer not to have spaces in the type and
> the member, makes it easier for my eyes.
>
Hmm. will fix.

> > + * RES_BELOW_LIMIT -- usage is smaller than limt, success.
>
>   ^^^^ typo
>
sure, will fix.

> > + spin_unlock_irqrestore(&counter->lock, flags);
> > + return ret;
> > +}
> > +
>
> When do we return RES_OVER_LIMIT? Are we missing that here?
>
It's Bug. Yamamoto-san pointed out and I'm now fixing.

> >   spin_lock_irqsave(&counter->lock, flags);
> > + switch (member) {
> > + case RES_LIMIT:
> > +   if (counter->use_watermark && counter->hwmark > tmp)
> > +   goto unlock_free;
>
> We need to document such API changes in the Documentation/controllers/memory.txt
> file.
>
ok, I'll add patch for documentation. to memory.txt and res_counter.txt.

> > +   break;
> > + case RES_HWMARK:
> > +   if (tmp < counter->lwmark || tmp > counter->limit)
> > +   goto unlock_free;
> > +   break;
> > + case RES_LWMARK:
> > +   if (tmp > counter->hwmark)
> > +   goto unlock_free;
> > +   break;
> > + default:

> > +   break;
> > + }
> >   val = res_counter_member(counter, member);
> >   *val = tmp;
> > - spin_unlock_irqrestore(&counter->lock, flags);
> >   ret = nbytes;
> > +unlock_free:
> > + spin_unlock_irqrestore(&counter->lock, flags);
> > out_free:
> >   kfree(buf);
> > out:
> > Index: mm-2.6.26-rc2-mm1/mm/memcontrol.c
> > ===
> > --- mm-2.6.26-rc2-mm1.orig/mm/memcontrol.c
> > +++ mm-2.6.26-rc2-mm1/mm/memcontrol.c
> > @@ -559,7 +559,7 @@ static int mem_cgroup_charge_common(stru
> >   css_get(&memcg->css);
> >   }
> >
> > - while (res_counter_charge(&mem->res, PAGE_SIZE)) {
> > + while (res_counter_charge(&mem->res, PAGE_SIZE) == RES_OVER_LIMIT) {
> >   if (!(gfp_mask & __GFP_WAIT))
> >   goto out;
> >
>
> Otherwise looks good so far. Need to look at the background reclaim code.
>

Thanks,
-Kame

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

---

Subject: Re: [RFC 4/4] memcg: NUMA background reclaim
Posted by KAMEZAWA Hiroyuki on Wed, 28 May 2008 00:44:23 GMT
View Forum Message <> Reply to Message

On Tue, 27 May 2008 22:56:43 +0530
Balbir Singh <balbir@linux.vnet.ibm.com> wrote:

> KAMEZAWA Hiroyuki wrote:
> > One aspect of difference in reclaim logic between global lru and memcg is
> > * global LRU triggers memory reclaim at memory shortage.
> > * memcg LRU triggers memory reclaim at excess of usage.

> >
> > Then, global LRU _know_ which node we should start reclaim from.
> > * start from a node at memory shortage or
> > * start from a node where memory allocation is waiting
> >
> > WRT memcg, it's difficult to find where we should start because
> > there is no memory shortage and LRU is splitted.
> > (But per-zone-LRU is definitely necessary for scalability.)
> >
> > This patch tries to deteremine a node for starting recalim by checking
> > ratio of inactive pages/active pages in a node. And trying to avoid starting
> > from a node with relatively small usage.
> > Better algorithm is welcome.
> >
> > Singed-off-by: KAMEZAWA Hiruyuki <kamezawa.hiroyu@jp.fujitsu.com>
> >
> > Index: mm-2.6.26-rc2-mm1/mm/memcontrol.c
> > ===
> > --- mm-2.6.26-rc2-mm1.orig/mm/memcontrol.c
> > +++ mm-2.6.26-rc2-mm1/mm/memcontrol.c
> > @@ -578,7 +578,7 @@ retry:
> >   if (state == RES_OVER_LIMIT) {
> >   if (!(gfp_mask & __GFP_WAIT))
> >   goto out;
> > - if (try_to_free_mem_cgroup_pages(mem, gfp_mask))
> > + if (try_to_free_mem_cgroup_pages(mem, -1, gfp_mask))
> >   goto retry;
> >   /*
> >   * try_to_free_mem_cgroup_pages() might not give us a
> > @@ -801,7 +801,7 @@ int mem_cgroup_shrink_usage(struct mm_st
> >   rcu_read_unlock();
> >
> >   do {
> > - progress = try_to_free_mem_cgroup_pages(mem, gfp_mask);
> > + progress = try_to_free_mem_cgroup_pages(mem, -1, gfp_mask);
> >   } while (!progress && --retry);
> >
> >   if (!retry)
> > @@ -814,7 +814,7 @@ static void mem_cgroup_drop_all_pages(st
> > {
> >   int progress;
> >   while (!res_counter_empty(&mem->res)) {
> > - progress = try_to_free_mem_cgroup_pages(mem,
> > + progress = try_to_free_mem_cgroup_pages(mem, -1,
> >   GFP_HIGHUSER_MOVABLE);
> >   if (!progress) /* we did as much as possible */
> >   break;
> > @@ -912,6 +912,62 @@ out:

```
> >  /*
> >   * background reclaim daemon.
> >   */
> > +
> > +#ifdef CONFIG_NUMA
> > +/*
> > + * Because memory controller's memory reclaim doesn't come from memory shortage,
> > + * we cannot know which node should be reclaimed in an easy way.
> > + * This routine select a node with inactive pages to be a node for starting
> > + * scanning.
> > + */
> > +int __select_best_node(struct mem_cgroup *mem)
> > +{
> > + int nid;
> > + int best_node = -1;
> > + unsigned long highest_inactive_ratio = 0;
> > + unsigned long active, inactive, inactive_ratio, total, threshold, flags;
> > + struct mem_cgroup_per_zone *mz;
> > + int zid;
> > +
> > + /*
> > +  * When a node's memory usage is smaller than
> > +    * total_usage/num_of_node * 75%, we don't select the node
> > +  */
> > + total = mem->res.usage >> PAGE_SHIFT;
> > + threshold = (total / num_node_state(N_HIGH_MEMORY)) * 3 / 4;
> > +
> > + /*
> > +  * See nodemask.h, N_HIGH_MEMORY means that a node has memory
> > +  * can be used for user's memory.(i.e. not means HIGHMEM).
> > +  */
> > + for_each_node_state(nid, N_HIGH_MEMORY) {
> > +  active = 0;
> > +  inactive = 0;
> > +
> > +  for (zid = 0; zid < MAX_NR_ZONES; zid++) {
> > +   mz = mem_cgroup_zoneinfo(mem, nid, zid);
> > +   spin_lock_irqsave(&mz->lru_lock, flags);
> > +   active += MEM_CGROUP_ZSTAT(mz, MEM_CGROUP_ZSTAT_ACTIVE);
> > +   inactive +=
> > +    MEM_CGROUP_ZSTAT(mz, MEM_CGROUP_ZSTAT_INACTIVE);
> > +   spin_unlock_irqrestore(&mz->lru_lock, flags);
> > +  }
> > +
> > +  if (active + inactive < threshold)
> > +   continue;
> > +  inactive_ratio = (inactive * 100) / (active + 1);
> > +  if (inactive_ratio > highest_inactive_ratio)
```

> > +   best_node = nid;
>
> Shouldn't we update highest_inactive_ration here?
>
AH, yes. blame me :(Thanks!

Thanks,
-Kame

_____
Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers

---

Subject: Re: [RFC 3/4] memcg: background reclaim
Posted by KAMEZAWA Hiroyuki on Wed, 28 May 2008 00:45:30 GMT
View Forum Message <> Reply to Message

On Tue, 27 May 2008 22:38:09 +0530
Balbir Singh <balbir@linux.vnet.ibm.com> wrote:
> > struct cgroup_subsys mem_cgroup_subsys __read_mostly;
> > @@ -119,10 +120,6 @@ struct mem_cgroup_lru_info {
> >   * statistics based on the statistics developed by Rik Van Riel for clock-pro,
> >   * to help the administrator determine what knobs to tune.
> >   *
> > - * TODO: Add a water mark for the memory controller. Reclaim will begin when
> > - * we hit the water mark. May be even add a low water mark, such that
> > - * no reclaim occurs from a cgroup at it's low water mark, this is
> > - * a feature that will be implemented much later in the future.
> >   */
> > struct mem_cgroup {
> >   struct cgroup_subsys_state css;
> > @@ -131,6 +128,13 @@ struct mem_cgroup {
> >   */
> >   struct res_counter res;
> >   /*
> > + * background reclaim.
> > + */
> > + struct {
> > + wait_queue_head_t waitq;
> > + struct task_struct *thread;
> > + } daemon;
>
> Comments on each of the members would be nice.
>
ok, I'll add.

> > + /*
> > * Per cgroup active and inactive list, similar to the
> > * per zone LRU lists.
> > */
> > @@ -143,6 +147,7 @@ struct mem_cgroup {
> > struct mem_cgroup_stat stat;
> > };
> > static struct mem_cgroup init_mem_cgroup;
> > +static DEFINE_MUTEX(memcont_daemon_lock);
> >
> > /*
> > * We use the lower bit of the page->page_cgroup pointer as a bit spin
> > @@ -374,6 +379,15 @@ void mem_cgroup_move_lists(struct page *
> > unlock_page_cgroup(page);
> > }
> >
> > +static void mem_cgroup_schedule_reclaim(struct mem_cgroup *mem)
> > +{
> > + if (!mem->daemon.thread)
> > + return;
>
> I suspect we are using threads. Aren't workqueues better than threads?
>
Hmm, I don't like to use generic workqueue for something which can take
long time and can _sleep_. How about using a thread for the whole system
for making service to all memcg ?

> > + if (!waitqueue_active(&mem->daemon.waitq))
> > + return;
> > + wake_up_interruptible(&mem->daemon.waitq);
> > +}
> > +
> > /*
> > * Calculate mapped_ratio under memory controller. This will be used in
> > * vmscan.c for deteremining we have to reclaim mapped pages.
> > @@ -532,6 +546,7 @@ static int mem_cgroup_charge_common(stru
> > unsigned long flags;
> > unsigned long nr_retries = MEM_CGROUP_RECLAIM_RETRIES;
> > struct mem_cgroup_per_zone *mz;
> > + enum res_state state;
> >
> > if (mem_cgroup_subsys.disabled)
> > return 0;
> > @@ -558,23 +573,23 @@ static int mem_cgroup_charge_common(stru
> > mem = memcg;

```
> >    css_get(&memcg->css);
> >   }
> > -
> > - while (res_counter_charge(&mem->res, PAGE_SIZE) == RES_OVER_LIMIT) {
> > +retry:
> > + state = res_counter_charge(&mem->res, PAGE_SIZE);
> > + if (state == RES_OVER_LIMIT) {
> >    if (!(gfp_mask & __GFP_WAIT))
> >     goto out;
> > -
> >    if (try_to_free_mem_cgroup_pages(mem, gfp_mask))
> > -   continue;
> > -
> > +   goto retry;
> >    /*
> > -   * try_to_free_mem_cgroup_pages() might not give us a full
> > -   * picture of reclaim. Some pages are reclaimed and might be
> > -   * moved to swap cache or just unmapped from the cgroup.
> > -   * Check the limit again to see if the reclaim reduced the
> > -   * current usage of the cgroup before giving up
> > +   * try_to_free_mem_cgroup_pages() might not give us a
> > +   * full picture of reclaim. Some pages are reclaimed
> > +   * and might be moved to swap cache or just unmapped
> > +   * from the cgroup. Check the limit again to see if
> > +   * the reclaim reduced the current usage of the cgroup
> > +   * before giving up
> >     */
> >    if (res_counter_check_under_limit(&mem->res))
> > -   continue;
> > +   goto retry;
> >
> >    if (!nr_retries--) {
> >     mem_cgroup_out_of_memory(mem, gfp_mask);
> > @@ -609,6 +624,9 @@ static int mem_cgroup_charge_common(stru
> >   spin_unlock_irqrestore(&mz->lru_lock, flags);
> >
> >   unlock_page_cgroup(page);
> > +
> > + if (state > RES_BELOW_HIGH)
> > +  mem_cgroup_schedule_reclaim(mem);
>
> I really don't like state > RES_BELOW_HIGH sort of checks. Could we please
> abstract them into functions, like
>
> if (mem_res_above_high_watermark(state))
>  ....
>
Hmm...ok.
```

> > done:
> >   return 0;
> > out:
> > @@ -891,6 +909,74 @@ out:
> >   css_put(&mem->css);
> >   return ret;
> > }
> > +/*
> > + * background reclaim daemon.
> > + */
> > +static int mem_cgroup_reclaim_daemon(void *data)
> > +{
> > + DEFINE_WAIT(wait);
> > + struct mem_cgroup *mem = data;
> > + int ret;
> > +
> > + css_get(&mem->css);
> > + current->flags |= PF_SWAPWRITE;
> > + /* we don't want to use cpu too much. */
> > + set_user_nice(current, 0);
>
> Shouldn't this (0) be a #define, what if we would like to degrade our nice value
> even further later?
>
Hmm, I think bare '0' is not harmful here. But ok, I'll add macro.

> > + set_freezable();
> > +
> > + while (!kthread_should_stop()) {
> > + prepare_to_wait(&mem->daemon.waitq, &wait, TASK_INTERRUPTIBLE);
> > + if (res_counter_state(&mem->res) == RES_BELOW_LOW) {
> > +   if (!kthread_should_stop()) {
> > +   schedule();
> > +   try_to_freeze();
>
> I am afraid, I fail to understand the code above.
>

kthread_should_stop() for checking 'we should exit this loop because someone
called rmdir() and kthread_stop() is issued.'

we tie 'current' to waitqueue and call schedule() if usage is lower than lwmark.

try_to_freeze() for checking 'we should stop here for a while because freezer
requires it'. See freeze_process() in kernel/power/freezer.c or some daemon

like pdflush().

> > + }
> > + finish_wait(&mem->daemon.waitq, &wait);
> > + continue;
> > + }
> > + finish_wait(&mem->daemon.waitq, &wait);
> > + /*
> > + * memory resource controller doesn't see NUMA memory usage
> > + * balancing, becasue we cannot know what balancing is good.
>
>   ^^^^^^^^ typo
>
ouch, thanks,

> > + * TODO: some annotation or heuristics to detect which node
> > + * we should start reclaim from.
> > + */
> > + ret = try_to_free_mem_cgroup_pages(mem, GFP_HIGHUSER_MOVABLE);
> > +
> > + yield();
>
> Why do we yeild here? Shouldn't our waitqueue handle the waiting?
>
just stopping to use cpu too much. One iteration of try_to_free_... is heavy
job. There are 2 reasons.
1. memcg can work well without bgreclaim...just relatively slow.
2. In following case,
  lwmark.....hwmark...limit
this kthread runs very long and moderate operation is required.

I can't catch "Shouldn't our waitqueue handle the waiting?" but
waitqueue check waitqueue is empty or not. caller of wakeup
don't have to take care of this.

> > + }
> > + css_put(&mem->css);
> > + return 0;
> > +}
> > +
> > +static int mem_cgroup_start_daemon(struct mem_cgroup *mem)
> > +{
> > + int ret = 0;
> > + struct task_struct *thr;
> > +

> > + mutex_lock(&memcont_daemon_lock);
> > + if (!mem->daemon.thread) {
> > + thr = kthread_run(mem_cgroup_reclaim_daemon, mem, "memcontd");
> > + if (IS_ERR(thr))
> > +   ret = PTR_ERR(thr);
> > + else
> > +   mem->daemon.thread = thr;
> > + }
> > + mutex_unlock(&memcont_daemon_lock);
> > + return ret;
> > +}
> > +
> > +static void mem_cgroup_stop_daemon(struct mem_cgroup *mem)
> > +{
> > + mutex_lock(&memcont_daemon_lock);
> > + if (mem->daemon.thread) {
> > + kthread_stop(mem->daemon.thread);
> > + mem->daemon.thread = NULL;
> > + }
> > + mutex_unlock(&memcont_daemon_lock);
> > + return;
> > +}
> > +
> >
> > static int mem_cgroup_write_strategy(char *buf, unsigned long long *tmp)
> > {
> > @@ -915,6 +1001,19 @@ static ssize_t mem_cgroup_write(struct c
> >   struct file *file, const char __user *userbuf,
> >   size_t nbytes, loff_t *ppos)
> > {
> > + int ret;
> > + /*
> > + * start daemon can fail. But we should start daemon always
> > + * when changes to HWMARK is succeeded. So, we start daemon before
> > + * changes to HWMARK. We don't stop this daemon even if
> > + * res_counter_write fails. To do that, we need ugly codes and
> > + * it's not so big problem.
> > + */
> > + if (cft->private == RES_HWMARK) {
> > + ret = mem_cgroup_start_daemon(mem_cgroup_from_cont(cont));
> > + if (ret)
> > +   return ret;
> > + }
> >   return res_counter_write(&mem_cgroup_from_cont(cont)->res,
> >   cft->private, userbuf, nbytes, ppos,
> >   mem_cgroup_write_strategy);
> > @@ -1004,6 +1103,18 @@ static struct cftype mem_cgroup_files[]
> >   .read_u64 = mem_cgroup_read,

> > },
> > {
> > + .name = "high_wmark_in_bytes",
> > + .private = RES_HWMARK,
> > + .write = mem_cgroup_write,
> > + .read_u64 = mem_cgroup_read,
> > + },
> > + {
> > + .name = "low_wmark_in_bytes",
> > + .private = RES_LWMARK,
> > + .write = mem_cgroup_write,
> > + .read_u64 = mem_cgroup_read,
> > + },
> > + {
> >   .name = "force_empty",
> >   .trigger = mem_force_empty_write,
> > },
> > @@ -1087,7 +1198,8 @@ mem_cgroup_create(struct cgroup_subsys *
> >   return ERR_PTR(-ENOMEM);
> > }
> >
> > - res_counter_init(&mem->res);
> > + res_counter_init_wmark(&mem->res);
> > + init_waitqueue_head(&mem->daemon.waitq);
> >
> > for_each_node_state(node, N_POSSIBLE)
> >   if (alloc_mem_cgroup_per_zone_info(mem, node))
> > @@ -1106,6 +1218,7 @@ static void mem_cgroup_pre_destroy(struc
> >   struct cgroup *cont)
> > {
> > struct mem_cgroup *mem = mem_cgroup_from_cont(cont);
> > + mem_cgroup_stop_daemon(mem);
> > mem_cgroup_force_empty(mem);
> > }
> >
>
> I failed to see code for the low watermark. Shouldn't we stop the thread once we
> go below the low watermark?
>
>
It stops. here.

> > + if (res_counter_state(&mem->res) == RES_BELOW_LOW) {
> > +   if (!kthread_should_stop()) {
> > +   schedule();

Thanks,

-Kame

_____

Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers