Subject: [RFC] memory controller : backgorund reclaim and avoid excessive locking [0/5]
Posted by KAMEZAWA Hiroyuki on Thu, 14 Feb 2008 08:21:48 GMT
View Forum Message <> Reply to Message

Hi, these are patches on my stack, now.

I believe test is not enough. Maybe there is BUG.

Before updating patches, I'd like to hear guys' advice against my concept
before going further.

TODO:
 - test on NUMA (machine is not avilable in these days. but I will do)

This series inludes 2 series of patches.
One is updated "back-ground reclaiming with high-low watermark".
One is for avoid locking.

Updated series.
[1/5] -- high-low watermark for resource counter.
[2/5] -- background reclaim with high-low watermark.
[3/5] -- throttle # of direct reclaim

This series tries to improve page reclaiming.

New ones.
[4/5] -- borrow resource for avoiding counter->lock
[5/5] -- bulk freeing page_cgroup

This series tries to reduce contention on counter->lock and mz->lru_lock.
This 2 are the top most contended lock in lock_stat.

Any commetns are welcome.

Regards,
-Kame

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

Subject: [RFC] memory controller : backgorund reclaim and avoid excessive locking [2/5] background reclaim.
Posted by KAMEZAWA Hiroyuki on Thu, 14 Feb 2008 08:27:40 GMT

A patch for background reclaim based on high-low watermak in res_counter.
The daemon is called as "memcontd", here.

Implements following:
 * If res->usage is higher than res->hwmark, start memcontd.
 * memcontd calls try_to_free_pages.
 * memcontd stops if res->usage is lower than res->lwmark.

Maybe we can add more tunings but no extra params now.

ChangeLog:
 - start "memcontd" at first change in hwmark.
   (In old verion, it started at cgroup creation.)
 - changed "relax" logic in memcontd daemon.

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

 mm/memcontrol.c | 112
++--
 1 files changed, 109 insertions(+), 3 deletions(-)

Index: linux-2.6.24-mm1/mm/memcontrol.c
===
--- linux-2.6.24-mm1.orig/mm/memcontrol.c
+++ linux-2.6.24-mm1/mm/memcontrol.c
@@ -30,6 +30,8 @@
 #include <linux/spinlock.h>
 #include <linux/fs.h>
 #include <linux/seq_file.h>
+#include <linux/kthread.h>
+#include <linux/freezer.h>

 #include <asm/uaccess.h>

@@ -136,6 +138,13 @@ struct mem_cgroup {
   * statistics.
   */
  struct mem_cgroup_stat stat;
+ /*
+ * background reclaim.
+ */
+ struct {
+ wait_queue_head_t waitq;
+ struct task_struct *kthread;
+ } daemon;
 };

```
 /*
@@ -504,6 +513,14 @@ long mem_cgroup_calc_reclaim_inactive(st
  return (nr_inactive >> priority);
 }

+static inline void mem_cgroup_schedule_daemon(struct mem_cgroup *mem)
+{
+ if (likely(mem->daemon.kthread) && /* can be NULL at boot */
+     waitqueue_active(&mem->daemon.waitq))
+  wake_up_interruptible(&mem->daemon.waitq);
+}
+
+
 unsigned long mem_cgroup_isolate_pages(unsigned long nr_to_scan,
       struct list_head *dst,
       unsigned long *scanned, int order,
@@ -658,6 +675,9 @@ retry:
   congestion_wait(WRITE, HZ/10);
  }

+ if (res_counter_above_hwmark(&mem->res))
+  mem_cgroup_schedule_daemon(mem);
+
  atomic_set(&pc->ref_cnt, 1);
  pc->mem_cgroup = mem;
  pc->page = page;
@@ -762,6 +782,50 @@ void mem_cgroup_uncharge_page(struct pag
 }

 /*
+ * background page reclaim routine for cgroup.
+ */
+static int mem_cgroup_reclaim_daemon(void *data)
+{
+ DEFINE_WAIT(wait);
+ struct mem_cgroup *mem = data;
+
+ css_get(&mem->css);
+ current->flags |= PF_SWAPWRITE;
+ set_freezable();
+
+ while (!kthread_should_stop()) {
+  prepare_to_wait(&mem->daemon.waitq, &wait, TASK_INTERRUPTIBLE);
+  if (res_counter_below_lwmark(&mem->res)) {
+   if (!kthread_should_stop()) {
+    schedule();
+    try_to_freeze();
+   }
```

```
+   finish_wait(&mem->daemon.waitq, &wait);
+   continue;
+  }
+  finish_wait(&mem->daemon.waitq, &wait);
+  try_to_free_mem_cgroup_pages(mem, GFP_HIGHUSER_MOVABLE);
+  /* Am I in hurry ? */
+  if (!res_counter_above_hwmark(&mem->res)) {
+   /*
+    * Extra relaxing..memory reclaim is hevay work.
+    * we don't know there is I/O congestion or not.
+    * So use just relax rather than congesiton_wait().
+    * HZ/10 is widely used value under /mm.
+    */
+   schedule_timeout(HZ/10);
+  } else {
+   /* Avoid occupation */
+   yield();
+  }
+ }
+
+ css_put(&mem->css);
+ return 0;
+}
+
+
+/*
  * Returns non-zero if a page (under migration) has valid page_cgroup member.
  * Refcnt of page_cgroup is incremented.
  */
@@ -931,15 +995,40 @@ static ssize_t mem_cgroup_read(struct cg
     NULL);
 }

+static DEFINE_MUTEX(modify_param_mutex);
 static ssize_t mem_cgroup_write(struct cgroup *cont, struct cftype *cft,
     struct file *file, const char __user *userbuf,
     size_t nbytes, loff_t *ppos)
 {
- return res_counter_write(&mem_cgroup_from_cont(cont)->res,
-    cft->private, userbuf, nbytes, ppos,
+ int ret;
+ struct mem_cgroup *mem = mem_cgroup_from_cont(cont);
+
+ mutex_lock(&modify_param_mutex);
+ /* Attach new background reclaim daemon.
+    This must be done before change values (for easy error handling */
+
+ if (cft->private == RES_HWMARK &&
```

```
+     !mem->daemon.kthread) {
+  struct task_struct *thr;
+  thr = kthread_run(mem_cgroup_reclaim_daemon, mem, "memcontd");
+  if (IS_ERR(thr)) {
+   ret = PTR_ERR(thr);
+   goto out;
+  }
+  mem->daemon.kthread = thr;
+ }
+ ret =  res_counter_write(&mem->res, cft->private, userbuf, nbytes, ppos,
     mem_cgroup_write_strategy);
+
+ /* Even on error, don't stop reclaim daemon here. not so problematic. */
+
+out:
+ mutex_unlock(&modify_param_mutex);
+ return ret;
 }

+
+
 static ssize_t mem_force_empty_write(struct cgroup *cont,
     struct cftype *cft, struct file *file,
     const char __user *userbuf,
@@ -1032,6 +1121,20 @@ static struct cftype mem_cgroup_files[]
   .write = mem_cgroup_write,
   .read = mem_cgroup_read,
  },
+
+ {
+  .name = "lwmark_in_bytes",
+  .private = RES_LWMARK,
+  .write = mem_cgroup_write,
+  .read = mem_cgroup_read,
+ },
+ {
+  .name = "hwmark_in_bytes",
+  .private = RES_HWMARK,
+  .write = mem_cgroup_write,
+  .read = mem_cgroup_read,
+ },
+
  {
   .name = "failcnt",
   .private = RES_FAILCNT,
@@ -1110,7 +1213,8 @@ mem_cgroup_create(struct cgroup_subsys *
  for_each_node_state(node, N_POSSIBLE)
   if (alloc_mem_cgroup_per_zone_info(mem, node))
```

```
    goto free_out;
-
+ init_waitqueue_head(&mem->daemon.waitq);
+ mem->daemon.kthread = NULL;
  return &mem->css;
 free_out:
  for_each_node_state(node, N_POSSIBLE)
@@ -1125,6 +1229,8 @@ static void mem_cgroup_pre_destroy(struc
 {
  struct mem_cgroup *mem = mem_cgroup_from_cont(cont);
  mem_cgroup_force_empty(mem);
+ if (mem->daemon.kthread)
+  kthread_stop(mem->daemon.kthread);
 }

 static void mem_cgroup_destroy(struct cgroup_subsys *ss,
```

Subject: [RFC] memory controller : backgorund reclaim and avoid excessive locking [1/5] high-low watermark
Posted by KAMEZAWA Hiroyuki on Thu, 14 Feb 2008 08:28:01 GMT
View Forum Message <> Reply to Message

Add High/Low watermark support to res_counter.

This adds high/low watermark support to res_counter.

Adds new member hwmark, lwmark, wmark_state to res_counter struct.

Users of res_counter must keep lwmark <= hwmark <= limit.
(set params as lwmark == hwmark == limit will disable this feature.)

wmark_state is added to read wmark status without lock.

Codes will be like this. (check status after charge.)

```
 if (res_counter_charge(cnt...)) { <----(need lock)
  .....
 }
 if (res_counter_above_hwmark(cnt)) <---- (no lock)
  trigger background jobs...
```

If I have to check under lock, please teach me.

counter->hwmark and counter->lwmark is not automatically adjusted
when limit is changed. So, users must change lwmark, hwmark before
changing limit.

Changelog
  * made variable names shorter.
  * adjusted to 2.6.24-mm1

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

 include/linux/res_counter.h | 30 ++++++++++++++++++++++++++
 kernel/res_counter.c | 53 ++-----
 2 files changed, 78 insertions(+), 5 deletions(-)

Index: linux-2.6.24-mm1/include/linux/res_counter.h
===
--- linux-2.6.24-mm1.orig/include/linux/res_counter.h
+++ linux-2.6.24-mm1/include/linux/res_counter.h
@@ -19,6 +19,16 @@
  * the helpers described beyond
  */

+/*
+ * Watermark status.
+ */
+
+enum watermark_state {
+ RES_WMARK_BELOW_LOW = 0, /* usage < low <= high <= max */
+ RES_WMARK_ABOVE_LOW, /* low <= usage < high <= max */
+ RES_WMARK_ABOVE_HIGH, /* low <= high <= usage <= max */
+};
+
 struct res_counter {
  /*
   * the current resource consumption level
@@ -33,10 +43,18 @@ struct res_counter {
   */
  unsigned long long failcnt;
  /*
+ * for supporting High/Low watermak
+ * Must keep low <= high <= limit.
+ */
+ unsigned long long hwmark;
+ unsigned long long lwmark;
+ enum watermark_state wmark_state; /* changed at charge/uncharge */
+ /*
   * the lock to protect all of the above.

```
   * the routines below consider this to be IRQ-safe
   */
  spinlock_t lock;
+
 };

 /*
@@ -66,6 +84,8 @@ enum {
  RES_USAGE,
  RES_LIMIT,
  RES_FAILCNT,
+ RES_HWMARK,
+ RES_LWMARK,
 };

 /*
@@ -124,4 +144,14 @@ static inline bool res_counter_check_und
  return ret;
 }

+static inline bool res_counter_below_lwmark(struct res_counter *cnt)
+{
+ smp_rmb();
+ return (cnt->wmark_state == RES_WMARK_BELOW_LOW);
+}
+static inline bool res_counter_above_hwmark(struct res_counter *cnt)
+{
+ smp_rmb();
+ return (cnt->wmark_state == RES_WMARK_ABOVE_HIGH);
+}
 #endif
Index: linux-2.6.24-mm1/kernel/res_counter.c
===================================================================
--- linux-2.6.24-mm1.orig/kernel/res_counter.c
+++ linux-2.6.24-mm1/kernel/res_counter.c
@@ -17,16 +17,29 @@ void res_counter_init(struct res_counter
 {
  spin_lock_init(&counter->lock);
  counter->limit = (unsigned long long)LLONG_MAX;
+ counter->lwmark = (unsigned long long)LLONG_MAX;
+ counter->hwmark = (unsigned long long)LLONG_MAX;
+ counter->wmark_state = RES_WMARK_BELOW_LOW;
 }

 int res_counter_charge_locked(struct res_counter *counter, unsigned long val)
 {
- if (counter->usage + val > counter->limit) {
+ unsigned long long newval = counter->usage + val;
```

```
+ if (newval > counter->limit) {
   counter->failcnt++;
   return -ENOMEM;
  }

- counter->usage += val;
+        if (newval > counter->hwmark) {
+  counter->wmark_state = RES_WMARK_ABOVE_HIGH;
+  smp_wmb();
+ } else if (newval > counter->lwmark) {
+  counter->wmark_state = RES_WMARK_ABOVE_LOW;
+  smp_wmb();
+ }
+
+ counter->usage = newval;
+
  return 0;
 }

@@ -43,10 +56,18 @@ int res_counter_charge(struct res_counte

 void res_counter_uncharge_locked(struct res_counter *counter, unsigned long val)
 {
+ unsigned long long newval = counter->usage - val;
  if (WARN_ON(counter->usage < val))
-  val = counter->usage;
+  newval = 0;

- counter->usage -= val;
+ if (newval < counter->lwmark) {
+  counter->wmark_state = RES_WMARK_BELOW_LOW;
+  smp_wmb();
+ } else if (newval < counter->hwmark) {
+  counter->wmark_state = RES_WMARK_ABOVE_LOW;
+  smp_wmb();
+ }
+ counter->usage = newval;
 }

 void res_counter_uncharge(struct res_counter *counter, unsigned long val)
@@ -69,6 +90,10 @@ 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();
@@ -123,10 +148,28 @@ ssize_t res_counter_write(struct res_cou
   goto out_free;
 }
 spin_lock_irqsave(&counter->lock, flags);
+ switch (member) {
+ case RES_LIMIT:
+  if (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:
```

_____

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

---

Subject: [RFC] memory controller : backgorund reclaim and avoid excessive locking [3/5] throttling
Posted by KAMEZAWA Hiroyuki on Thu, 14 Feb 2008 08:30:43 GMT
View Forum Message <> Reply to Message

Throttle memory reclaim interface for cgroup.

This patch adds..

- a limit for simultaneous callers of try_to_free_mem_cgroup_pages().
- interface for that. memory.shrinkers.

There are some reasons.
- try_to_free... is very heavy and shoulnd't be called too much at once.
- When the number of callers of try_to_free.. is big, we'll reclaim
  too much memory.

By this interface, a user can control the # of threads which can enter
try_to_free...

Default is 10240 ...a enough big number for unlimited. Maybe this should
be changed.

Changes from previous one.
- Added an interface to control the limit.
- don't call wake_up at uncharge()...it seems hevay..
  Instead of that, sleepers use schedule_timeout(HZ/100). (10ms)

Considerations:
- Should we add this 'throttle' to global_lru at first ?
- Do we need more knobs ?
- Should default value to be automtically estimated value ?
  (I used '# of cpus/4' as default in previous version.)

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

```
 mm/memcontrol.c |  235 ++++++++++++++++++++++++++++++++++-----------------------------
 1 files changed, 127 insertions(+), 108 deletions(-)

Index: linux-2.6.24-mm1/mm/memcontrol.c
===================================================================
--- linux-2.6.24-mm1.orig/mm/memcontrol.c
+++ linux-2.6.24-mm1/mm/memcontrol.c
@@ -145,8 +145,17 @@ struct mem_cgroup {
   wait_queue_head_t waitq;
   struct task_struct *kthread;
  } daemon;
+ /*
+  * throttling params for reclaim.
+  */
+ struct {
+  int  limit;
+  atomic_t reclaimers;
+  wait_queue_head_t waitq;
```

```
+ } throttle;
 };

+
 /*
  * We use the lower bit of the page->page_cgroup pointer as a bit spin
  * lock. We need to ensure that page->page_cgroup is atleast two
@@ -520,6 +529,27 @@ static inline void mem_cgroup_schedule_d
   wake_up_interruptible(&mem->daemon.waitq);
 }

+static inline void mem_cgroup_wait_reclaim(struct mem_cgroup *mem)
+{
+ DEFINE_WAIT(wait);
+ while (1) {
+  prepare_to_wait(&mem->throttle.waitq, &wait,
+      TASK_INTERRUPTIBLE);
+  if (res_counter_check_under_limit(&mem->res)) {
+   finish_wait(&mem->throttle.waitq, &wait);
+   break;
+  }
+  /* expect some progress in... */
+  schedule_timeout(HZ/50);
+  finish_wait(&mem->throttle.waitq, &wait);
+ }
+}
+
+static inline int mem_cgroup_throttle_reclaim(struct mem_cgroup *mem)
+{
+ return atomic_add_unless(&mem->throttle.reclaimers, 1,
+     mem->throttle.limit);
+}

 unsigned long mem_cgroup_isolate_pages(unsigned long nr_to_scan,
      struct list_head *dst,
@@ -652,11 +682,22 @@ retry:
   * the cgroup limit.
   */
  while (res_counter_charge(&mem->res, PAGE_SIZE)) {
+  int ret;
   if (!(gfp_mask & __GFP_WAIT))
    goto out;

-  if (try_to_free_mem_cgroup_pages(mem, gfp_mask))
+  if (((gfp_mask & (__GFP_FS|__GFP_IO)) != (__GFP_FS|__GFP_IO))
+      || mem_cgroup_throttle_reclaim(mem)) {
+   ret = try_to_free_mem_cgroup_pages(mem, gfp_mask);
+   atomic_dec(&mem->throttle.reclaimers);
```

```
+   if (waitqueue_active(&mem->throttle.waitq))
+    wake_up_all(&mem->throttle.waitq);
+   if (ret)
+    continue;
+  } else {
+   mem_cgroup_wait_reclaim(mem);
    continue;
+  }

   /*
    * try_to_free_mem_cgroup_pages() might not give us a full
@@ -1054,6 +1095,19 @@ static ssize_t mem_force_empty_read(stru
  return -EINVAL;
 }

+static int mem_throttle_write(struct cgroup *cont, struct cftype *cft, u64 val)
+{
+ struct mem_cgroup *mem = mem_cgroup_from_cont(cont);
+ int limit = (int)val;
+ mem->throttle.limit = limit;
+ return 0;
+}
+
+static u64 mem_throttle_read(struct cgroup *cont, struct cftype *cft)
+{
+ struct mem_cgroup *mem = mem_cgroup_from_cont(cont);
+ return (u64)mem->throttle.limit;
+}

 static const struct mem_cgroup_stat_desc {
  const char *msg;
@@ -1146,6 +1200,11 @@ static struct cftype mem_cgroup_files[]
   .read = mem_force_empty_read,
  },
  {
+  .name = "shrinks",
+  .write_uint = mem_throttle_write,
+  .read_uint  = mem_throttle_read,
+ },
+ {
   .name = "stat",
   .open = mem_control_stat_open,
  },
@@ -1215,6 +1274,11 @@ mem_cgroup_create(struct cgroup_subsys *
    goto free_out;
  init_waitqueue_head(&mem->daemon.waitq);
  mem->daemon.kthread = NULL;
+
```

```
+ init_waitqueue_head(&mem->throttle.waitq);
+ mem->throttle.limit = 10240; /* maybe enough big for no throttle */
+ atomic_set(&mem->throttle.reclaimers, 0);
+
  return &mem->css;
 free_out:
  for_each_node_state(node, N_POSSIBLE)
```

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

---

Subject: Re: [RFC] memory controller : backgorund reclaim and avoid excessive locking [2/5] background reclai
Posted by Balbir Singh on Thu, 14 Feb 2008 08:32:44 GMT
View Forum Message <> Reply to Message

KAMEZAWA Hiroyuki wrote:
> A patch for background reclaim based on high-low watermak in res_counter.
> The daemon is called as "memcontd", here.
>
> Implements following:
> * If res->usage is higher than res->hwmark, start memcontd.
> * memcontd calls try_to_free_pages.
> * memcontd stops if res->usage is lower than res->lwmark.
>

Can we call the dameon memcgroupd?

> Maybe we can add more tunings but no extra params now.
>
> ChangeLog:
> - start "memcontd" at first change in hwmark.
>   (In old verion, it started at cgroup creation.)
> - changed "relax" logic in memcontd daemon.
>
> Signed-off-by: KAMEZAWA Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
>
> mm/memcontrol.c | 112
++--
> 1 files changed, 109 insertions(+), 3 deletions(-)
>
> Index: linux-2.6.24-mm1/mm/memcontrol.c
> ===
> --- linux-2.6.24-mm1.orig/mm/memcontrol.c
> +++ linux-2.6.24-mm1/mm/memcontrol.c

```
> @@ -30,6 +30,8 @@
>  #include <linux/spinlock.h>
>  #include <linux/fs.h>
>  #include <linux/seq_file.h>
> +#include <linux/kthread.h>
> +#include <linux/freezer.h>
>
>  #include <asm/uaccess.h>
>
> @@ -136,6 +138,13 @@ struct mem_cgroup {
>     * statistics.
>     */
>   struct mem_cgroup_stat stat;
> + /*
> +  * background reclaim.
> +  */
> + struct {
> +  wait_queue_head_t waitq;
> +  struct task_struct *kthread;
> + } daemon;
>  };
>
>  /*
> @@ -504,6 +513,14 @@ long mem_cgroup_calc_reclaim_inactive(st
>   return (nr_inactive >> priority);
>  }
>
> +static inline void mem_cgroup_schedule_daemon(struct mem_cgroup *mem)
> +{
> + if (likely(mem->daemon.kthread) && /* can be NULL at boot */
> +     waitqueue_active(&mem->daemon.waitq))
> +  wake_up_interruptible(&mem->daemon.waitq);
> +}
> +
> +
>  unsigned long mem_cgroup_isolate_pages(unsigned long nr_to_scan,
>       struct list_head *dst,
>       unsigned long *scanned, int order,
> @@ -658,6 +675,9 @@ retry:
>    congestion_wait(WRITE, HZ/10);
>   }
>
> + if (res_counter_above_hwmark(&mem->res))
> +  mem_cgroup_schedule_daemon(mem);
> +
```

I don't understand this part, some comments might be good here. Why do we wake up the dameon in mem_cgroup_isolate_pages? Why not do so under memory pressure

like the soft limit patches does? I suspect that these patches and soft limit patches might have some overlap.

```
>   atomic_set(&pc->ref_cnt, 1);
>   pc->mem_cgroup = mem;
>   pc->page = page;
> @@ -762,6 +782,50 @@ void mem_cgroup_uncharge_page(struct pag
>  }
>
>  /*
> + * background page reclaim routine for cgroup.
> + */
> +static int mem_cgroup_reclaim_daemon(void *data)
> +{
> + DEFINE_WAIT(wait);
> + struct mem_cgroup *mem = data;
> +
> + css_get(&mem->css);
> + current->flags |= PF_SWAPWRITE;
> + set_freezable();
> +
> + while (!kthread_should_stop()) {
> +  prepare_to_wait(&mem->daemon.waitq, &wait, TASK_INTERRUPTIBLE);
> +  if (res_counter_below_lwmark(&mem->res)) {
> +   if (!kthread_should_stop()) {
> +    schedule();
> +    try_to_freeze();
> +   }
> +   finish_wait(&mem->daemon.waitq, &wait);
> +   continue;
> +  }
> +  finish_wait(&mem->daemon.waitq, &wait);
> +  try_to_free_mem_cgroup_pages(mem, GFP_HIGHUSER_MOVABLE);
> +  /* Am I in hurry ? */
> +  if (!res_counter_above_hwmark(&mem->res)) {
> +   /*
> +    * Extra relaxing..memory reclaim is hevay work.
> +    * we don't know there is I/O congestion or not.
> +    * So use just relax rather than congesiton_wait().
> +    * HZ/10 is widely used value under /mm.
> +    */
> +   schedule_timeout(HZ/10);
```

Why not just yeild() here and let another iteration push us back to our low water mark?

```
> +  } else {
> +   /* Avoid occupation */
```

```
> +   yield();
> +  }
> + }
> +
> + css_put(&mem->css);
> + return 0;
> +}
> +
> +
> +/*
>   * Returns non-zero if a page (under migration) has valid page_cgroup member.
>   * Refcnt of page_cgroup is incremented.
>   */
> @@ -931,15 +995,40 @@ static ssize_t mem_cgroup_read(struct cg
>      NULL);
>  }
>
> +static DEFINE_MUTEX(modify_param_mutex);
>  static ssize_t mem_cgroup_write(struct cgroup *cont, struct cftype *cft,
>      struct file *file, const char __user *userbuf,
>      size_t nbytes, loff_t *ppos)
>  {
> - return res_counter_write(&mem_cgroup_from_cont(cont)->res,
> -    cft->private, userbuf, nbytes, ppos,
> + int ret;
> + struct mem_cgroup *mem = mem_cgroup_from_cont(cont);
> +
> + mutex_lock(&modify_param_mutex);
> + /* Attach new background reclaim daemon.
> +    This must be done before change values (for easy error handling */
> +
> + if (cft->private == RES_HWMARK &&
> +     !mem->daemon.kthread) {
> +  struct task_struct *thr;
> +  thr = kthread_run(mem_cgroup_reclaim_daemon, mem, "memcontd");
> +  if (IS_ERR(thr)) {
> +   ret = PTR_ERR(thr);
> +   goto out;
> +  }
> +  mem->daemon.kthread = thr;
> + }
> + ret =  res_counter_write(&mem->res, cft->private, userbuf, nbytes, ppos,
>      mem_cgroup_write_strategy);
> +
> + /* Even on error, don't stop reclaim daemon here. not so problematic. */
> +
> +out:
> + mutex_unlock(&modify_param_mutex);
```

> + return ret;
> }
>

Could we document the changes to this function. It looks like writing a value to the watermarks can cause the dameon to be scheduled.

> +
> +
> static ssize_t mem_force_empty_write(struct cgroup *cont,
>   struct cftype *cft, struct file *file,
>   const char __user *userbuf,
> @@ -1032,6 +1121,20 @@ static struct cftype mem_cgroup_files[]
>   .write = mem_cgroup_write,
>   .read = mem_cgroup_read,
>   },
> +
> + {
> + .name = "lwmark_in_bytes",
> + .private = RES_LWMARK,
> + .write = mem_cgroup_write,
> + .read = mem_cgroup_read,
> + },
> + {
> + .name = "hwmark_in_bytes",
> + .private = RES_HWMARK,
> + .write = mem_cgroup_write,
> + .read = mem_cgroup_read,
> + },
> +
>   {
>   .name = "failcnt",
>   .private = RES_FAILCNT,
> @@ -1110,7 +1213,8 @@ mem_cgroup_create(struct cgroup_subsys *
>   for_each_node_state(node, N_POSSIBLE)
>   if (alloc_mem_cgroup_per_zone_info(mem, node))
>   goto free_out;
> -
> + init_waitqueue_head(&mem->daemon.waitq);
> + mem->daemon.kthread = NULL;
>   return &mem->css;
> free_out:
>   for_each_node_state(node, N_POSSIBLE)
> @@ -1125,6 +1229,8 @@ static void mem_cgroup_pre_destroy(struc
> {
>   struct mem_cgroup *mem = mem_cgroup_from_cont(cont);
>   mem_cgroup_force_empty(mem);

> + if (mem->daemon.kthread)
> + kthread_stop(mem->daemon.kthread);
> }
>
> static void mem_cgroup_destroy(struct cgroup_subsys *ss,
>

--
 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: [RFC] memory controller : backgorund reclaim and avoid excessive locking [4/5] borrow resource
Posted by KAMEZAWA Hiroyuki on Thu, 14 Feb 2008 08:33:45 GMT
View Forum Message <> Reply to Message

One of contended lock is counter->lock.

Now, counter->usage is changed by PAGE_SIZE. This patch changes
this to be PAGE_SIZE * borrow_factor and cache "borrow" in
per cpu area.

This reduce # of lock against counter->lock.

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

Index: linux-2.6.24-mm1/mm/memcontrol.c
===
--- linux-2.6.24-mm1.orig/mm/memcontrol.c
+++ linux-2.6.24-mm1/mm/memcontrol.c
@@ -47,7 +47,8 @@ enum mem_cgroup_stat_index {
   */
  MEM_CGROUP_STAT_CACHE,   /* # of pages charged as cache */
  MEM_CGROUP_STAT_RSS,   /* # of pages charged as rss */
-
+ MEM_CGROUP_STAT_BORROW,   /* # of per-cpu borrow resource from
+   global resource */
  MEM_CGROUP_STAT_NSTATS,
 };

```
@@ -134,6 +135,9 @@ struct mem_cgroup {
  struct mem_cgroup_lru_info info;

  int prev_priority; /* for recording reclaim priority */
+
+ int borrow_unit;   /* size of unit for borrowing resource */
+
  /*
   * statistics.
   */
@@ -611,6 +615,92 @@ unsigned long mem_cgroup_isolate_pages(u
  return nr_taken;
 }

+/* FIXME?  we assume that size is always PAGE_SIZE. */
+
+static int mem_cgroup_borrow_and_charge(struct mem_cgroup *mem, int size)
+{
+ unsigned long flags;
+ int ret;
+
+ ret = 0;
+
+ local_irq_save(flags);
+ if (mem->borrow_unit) {
+  int cpu;
+  s64 *bwp;
+  cpu = smp_processor_id();
+  bwp = &mem->stat.cpustat[cpu].count[MEM_CGROUP_STAT_BORROW];
+  if (*bwp > size) {
+   *bwp -= size;
+   goto out;
+  }
+  /* try to charge */
+  ret = res_counter_charge(&mem->res, mem->borrow_unit);
+  if (!ret) { /* success */
+   *bwp += (mem->borrow_unit - size);
+   goto out;
+  }
+ }
+ spin_lock(&mem->res.lock);
+ ret = res_counter_charge_locked(&mem->res, size);
+ spin_unlock(&mem->res.lock);
+out:
+ local_irq_restore(flags);
+ return ret;
+}
```

```
+
+static void mem_cgroup_return_and_uncharge(struct mem_cgroup *mem, int size)
+{
+ unsigned long flags;
+ int uncharge_size = 0;
+
+ local_irq_save(flags);
+ if (mem->borrow_unit) {
+  int limit = mem->borrow_unit * 2;
+  int cpu;
+  s64 *bwp;
+  cpu = smp_processor_id();
+  bwp = &mem->stat.cpustat[cpu].count[MEM_CGROUP_STAT_BORROW];
+  *bwp += size;
+  if (*bwp > limit) {
+   uncharge_size = *bwp - mem->borrow_unit;
+   *bwp = mem->borrow_unit;
+  }
+ } else
+  uncharge_size = size;
+
+ if (uncharge_size) {
+  spin_lock(&mem->res.lock);
+  res_counter_uncharge_locked(&mem->res, size);
+  spin_unlock(&mem->res.lock);
+ }
+ local_irq_restore(flags);
+
+ return;
+}
+
+static void drain_local_borrow(void *data)
+{
+ int cpu;
+ int borrow;
+ unsigned long flags;
+ struct mem_cgroup *mem = data;
+
+ local_irq_save(flags);
+ cpu = smp_processor_id();
+ borrow = mem->stat.cpustat[cpu].count[MEM_CGROUP_STAT_BORROW];
+ mem->stat.cpustat[cpu].count[MEM_CGROUP_STAT_BORROW] = 0;
+ spin_lock(&mem->res.lock);
+ res_counter_uncharge_locked(&mem->res, borrow);
+ spin_unlock(&mem->res.lock);
+ local_irq_restore(flags);
+}
+
```

```
+static void drain_all_borrow(struct mem_cgroup *mem)
+{
+ on_each_cpu(drain_local_borrow, mem, 0, 1);
+}
+
 /*
  * Charge the memory controller for page usage.
  * Return
@@ -681,7 +771,7 @@ retry:
   * If we created the page_cgroup, we should free it on exceeding
   * the cgroup limit.
   */
- while (res_counter_charge(&mem->res, PAGE_SIZE)) {
+ while (mem_cgroup_borrow_and_charge(mem, PAGE_SIZE)) {
   int ret;
   if (!(gfp_mask & __GFP_WAIT))
    goto out;
@@ -709,6 +799,8 @@ retry:
   if (res_counter_check_under_limit(&mem->res))
    continue;

+  if (nr_retries < MEM_CGROUP_RECLAIM_RETRIES)
+   drain_all_borrow(mem);
   if (!nr_retries--) {
    mem_cgroup_out_of_memory(mem, gfp_mask);
    goto out;
@@ -805,7 +897,7 @@ void mem_cgroup_uncharge(struct page_cgr
   if (clear_page_cgroup(page, pc) == pc) {
    mem = pc->mem_cgroup;
    css_put(&mem->css);
-   res_counter_uncharge(&mem->res, PAGE_SIZE);
+   mem_cgroup_return_and_uncharge(mem, PAGE_SIZE);
    spin_lock_irqsave(&mz->lru_lock, flags);
    __mem_cgroup_remove_list(pc);
    spin_unlock_irqrestore(&mz->lru_lock, flags);
@@ -1005,6 +1097,7 @@ int mem_cgroup_force_empty(struct mem_cg
     /* drop all page_cgroup in inactive_list */
     mem_cgroup_force_empty_list(mem, mz, 0);
    }
+  drain_all_borrow(mem);
  }
  ret = 0;
 out:
@@ -1109,12 +1202,29 @@ static u64 mem_throttle_read(struct cgro
  return (u64)mem->throttle.limit;
 }

+static int mem_bulkratio_write(struct cgroup *cont, struct cftype *cft, u64 val)
```

```
+{
+ struct mem_cgroup *mem = mem_cgroup_from_cont(cont);
+ int unit = val * PAGE_SIZE;
+ if (unit > (PAGE_SIZE << (MAX_ORDER/2)))
+  return -EINVAL;
+ mem->borrow_unit = unit;
+ return 0;
+}
+
+static u64 mem_bulkratio_read(struct cgroup *cont, struct cftype *cft)
+{
+ struct mem_cgroup *mem = mem_cgroup_from_cont(cont);
+ return (u64)(mem->borrow_unit/PAGE_SIZE);
+}
+
 static const struct mem_cgroup_stat_desc {
  const char *msg;
  u64 unit;
 } mem_cgroup_stat_desc[] = {
  [MEM_CGROUP_STAT_CACHE] = { "cache", PAGE_SIZE, },
  [MEM_CGROUP_STAT_RSS] = { "rss", PAGE_SIZE, },
+ [MEM_CGROUP_STAT_BORROW] = { "borrow", 1, },
 };

 static int mem_control_stat_show(struct seq_file *m, void *arg)
@@ -1205,6 +1315,11 @@ static struct cftype mem_cgroup_files[]
   .read_uint  = mem_throttle_read,
  },
  {
+  .name = "bulkratio",
+  .write_uint = mem_bulkratio_write,
+  .read_uint = mem_bulkratio_read,
+ },
+ {
   .name = "stat",
   .open = mem_control_stat_open,
  },
@@ -1279,6 +1394,8 @@ mem_cgroup_create(struct cgroup_subsys *
  mem->throttle.limit = 10240; /* maybe enough big for no throttle */
  atomic_set(&mem->throttle.reclaimers, 0);

+ mem->borrow_unit = 0; /* Work at strict/precise mode as default */
+
  return &mem->css;
 free_out:
  for_each_node_state(node, N_POSSIBLE)
```

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

Subject: [RFC] memory controller : backgorund reclaim and avoid excessive locking [5/5] lazy page_cgroup free
Posted by KAMEZAWA Hiroyuki on Thu, 14 Feb 2008 08:36:24 GMT
View Forum Message <> Reply to Message

A big lock contetion of memory controller is mz->lru_lock.

This is acquired when
 1. add to lru
 2. remove from lru
 3. scan lru list

It seems 1. and 3. are unavoidable. but 2. can be delayed.

This patch make removing page_cgorup from lru-list be lazy and batched.
(Like pagevec..)

This patch adds a new flag page_cgroup and make lru scan routine
ignores it.

I think this reduces lock contention especially when
 - several tasks are exiting at once.
 - several files are removed at once.

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

```
 mm/memcontrol.c |   86 +++++++++++++++++++++++++++++++++++++++++++++++++++++++++++-------
 1 files changed, 76 insertions(+), 10 deletions(-)

Index: linux-2.6.24-mm1/mm/memcontrol.c
===================================================================
--- linux-2.6.24-mm1.orig/mm/memcontrol.c
+++ linux-2.6.24-mm1/mm/memcontrol.c
@@ -100,6 +100,7 @@ struct mem_cgroup_per_zone {
  struct list_head inactive_list;
  unsigned long count[NR_MEM_CGROUP_ZSTAT];
 };
+
 /* Macro for accessing counter */
 #define MEM_CGROUP_ZSTAT(mz, idx) ((mz)->count[(idx)])
```

```
@@ -157,9 +158,17 @@ struct mem_cgroup {
   atomic_t reclaimers;
   wait_queue_head_t waitq;
  } throttle;
+ /*
+  * For lazy freeing (not GC)
+  */
+ struct {
+  struct mem_cgroup_per_zone *mz;
+  int   num;
+#define GARBAGE_MAXSIZE  (16)
+  struct page_cgroup *vec[GARBAGE_MAXSIZE];
+ } garbage[NR_CPUS];
 };

-
 /*
  * We use the lower bit of the page->page_cgroup pointer as a bit spin
  * lock. We need to ensure that page->page_cgroup is atleast two
@@ -176,12 +185,14 @@ struct page_cgroup {
  struct list_head lru;  /* per cgroup LRU list */
  struct page *page;
  struct mem_cgroup *mem_cgroup;
+ struct mem_cgroup_per_zone *mz; /* now belongs to...*/
  atomic_t ref_cnt;  /* Helpful when pages move b/w  */
      /* mapped and cached states     */
  int  flags;
 };
 #define PAGE_CGROUP_FLAG_CACHE (0x1) /* charged as cache */
 #define PAGE_CGROUP_FLAG_ACTIVE (0x2) /* page is active in this cgroup */
+#define PAGE_CGROUP_FLAG_GARBAGE (0x4)  /* this page_cgroup is just a garbage */

 static inline int page_cgroup_nid(struct page_cgroup *pc)
 {
@@ -235,8 +246,12 @@ static inline struct mem_cgroup_per_zone
 page_cgroup_zoneinfo(struct page_cgroup *pc)
 {
  struct mem_cgroup *mem = pc->mem_cgroup;
- int nid = page_cgroup_nid(pc);
- int zid = page_cgroup_zid(pc);
+ int nid, zid;
+
+ if (pc->mz)
+  return pc->mz;
+ nid = page_cgroup_nid(pc);
+ zid = page_cgroup_zid(pc);

  return mem_cgroup_zoneinfo(mem, nid, zid);
```

```
 }
@@ -360,8 +375,10 @@ static struct page_cgroup *clear_page_cg
  /* lock and clear */
  lock_page_cgroup(page);
  ret = page_get_page_cgroup(page);
- if (likely(ret == pc))
+ if (likely(ret == pc)) {
   page_assign_page_cgroup(page, NULL);
+  pc->flags |= PAGE_CGROUP_FLAG_GARBAGE;
+ }
  unlock_page_cgroup(page);
  return ret;
 }
@@ -377,6 +394,7 @@ static void __mem_cgroup_remove_list(str
   MEM_CGROUP_ZSTAT(mz, MEM_CGROUP_ZSTAT_INACTIVE) -= 1;

  mem_cgroup_charge_statistics(pc->mem_cgroup, pc->flags, false);
+ pc->mz = NULL;
  list_del_init(&pc->lru);
 }

@@ -392,6 +410,7 @@ static void __mem_cgroup_add_list(struct
   MEM_CGROUP_ZSTAT(mz, MEM_CGROUP_ZSTAT_ACTIVE) += 1;
   list_add(&pc->lru, &mz->active_list);
  }
+ pc->mz = mz;
  mem_cgroup_charge_statistics(pc->mem_cgroup, pc->flags, true);
 }

@@ -408,10 +427,12 @@ static void __mem_cgroup_move_lists(stru
  if (active) {
   MEM_CGROUP_ZSTAT(mz, MEM_CGROUP_ZSTAT_ACTIVE) += 1;
   pc->flags |= PAGE_CGROUP_FLAG_ACTIVE;
+  pc->mz = mz;
   list_move(&pc->lru, &mz->active_list);
  } else {
   MEM_CGROUP_ZSTAT(mz, MEM_CGROUP_ZSTAT_INACTIVE) += 1;
   pc->flags &= ~PAGE_CGROUP_FLAG_ACTIVE;
+  pc->mz = mz;
   list_move(&pc->lru, &mz->inactive_list);
  }
 }
@@ -607,7 +628,6 @@ unsigned long mem_cgroup_isolate_pages(u
    nr_taken++;
   }
  }
-
  list_splice(&pc_list, src);
```

```
  spin_unlock(&mz->lru_lock);

@@ -703,6 +723,7 @@ static void drain_all_borrow(struct mem_
  on_each_cpu(drain_local_borrow, mem, 0, 1);
 }

+
 /*
  * Charge the memory controller for page usage.
  * Return
@@ -820,6 +841,7 @@ retry:
  pc->flags = PAGE_CGROUP_FLAG_ACTIVE;
  if (ctype == MEM_CGROUP_CHARGE_TYPE_CACHE)
   pc->flags |= PAGE_CGROUP_FLAG_CACHE;
+ pc->mz = NULL;

  if (!page || page_cgroup_assign_new_page_cgroup(page, pc)) {
   /*
@@ -872,6 +894,52 @@ int mem_cgroup_cache_charge(struct page
  return ret;
 }

+/* called under irq-off */
+void local_free_garbages(void *data)
+{
+ struct mem_cgroup *mem = data;
+ int cpu = smp_processor_id();
+ struct page_cgroup *pc, *tmp;
+ LIST_HEAD(frees);
+ int i;
+ if (!mem->garbage[cpu].mz)
+  return;
+ spin_lock(&mem->garbage[cpu].mz->lru_lock);
+ for (i = 0; i < mem->garbage[cpu].num; i++) {
+  pc = mem->garbage[cpu].vec[i];
+  __mem_cgroup_remove_list(pc);
+  list_add(&pc->lru, &frees);
+ }
+ mem->garbage[cpu].num = 0;
+ spin_unlock(&mem->garbage[cpu].mz->lru_lock);
+
+ list_for_each_entry_safe(pc, tmp, &frees, lru)
+  kfree(pc);
+}
+
+void __mem_cgroup_free_garbage(struct mem_cgroup *mem,
+    struct page_cgroup *pc)
+{
```

```
+ unsigned long flags;
+ int cpu;
+ local_irq_save(flags);
+ cpu = smp_processor_id();
+ if ((pc->mz != mem->garbage[cpu].mz) ||
+     (mem->garbage[cpu].num == GARBAGE_MAXSIZE))
+  local_free_garbages(mem);
+ mem->garbage[cpu].mz = pc->mz;
+ mem->garbage[cpu].vec[mem->garbage[cpu].num] = pc;
+ mem->garbage[cpu].num++;
+ local_irq_restore(flags);
+}
+
+void all_free_garbages(struct mem_cgroup *mem)
+{
+ on_each_cpu(local_free_garbages, mem, 0, 1);
+}
+
+
+
 /*
  * Uncharging is always a welcome operation, we never complain, simply
  * uncharge. This routine should be called with lock_page_cgroup held
@@ -881,7 +949,6 @@ void mem_cgroup_uncharge(struct page_cgr
  struct mem_cgroup *mem;
  struct mem_cgroup_per_zone *mz;
  struct page *page;
- unsigned long flags;

  /*
   * Check if our page_cgroup is valid
@@ -901,10 +968,7 @@ void mem_cgroup_uncharge(struct page_cgr
    mem = pc->mem_cgroup;
    css_put(&mem->css);
    mem_cgroup_return_and_uncharge(mem, PAGE_SIZE);
-   spin_lock_irqsave(&mz->lru_lock, flags);
-   __mem_cgroup_remove_list(pc);
-   spin_unlock_irqrestore(&mz->lru_lock, flags);
-   kfree(pc);
+    __mem_cgroup_free_garbage(mem, pc);
   }
   lock_page_cgroup(page);
  }
@@ -1050,11 +1114,15 @@ mem_cgroup_force_empty_list(struct mem_c
  if (list_empty(list))
   return;
 retry:
+ all_free_garbages(mem);
```

```
  count = FORCE_UNCHARGE_BATCH;
  spin_lock_irqsave(&mz->lru_lock, flags);

  while (--count && !list_empty(list)) {
   pc = list_entry(list->prev, struct page_cgroup, lru);
+  /* If there are still garbage, exit and retry */
+  if (pc->flags & PAGE_CGROUP_FLAG_GARBAGE)
+   break;
   page = pc->page;
   /* Avoid race with charge */
   atomic_set(&pc->ref_cnt, 0);
@@ -1102,6 +1170,7 @@ int mem_cgroup_force_empty(struct mem_cg
     mem_cgroup_force_empty_list(mem, mz, 0);
    }
   drain_all_borrow(mem);
+  all_free_garbages(mem);
  }
  ret = 0;
 out:
@@ -1416,6 +1485,7 @@ static void mem_cgroup_pre_destroy(struc
  mem_cgroup_force_empty(mem);
  if (mem->daemon.kthread)
   kthread_stop(mem->daemon.kthread);
+ all_free_garbages(mem);
 }

 static void mem_cgroup_destroy(struct cgroup_subsys *ss,
```

Subject: Re: [RFC] memory controller : backgorund reclaim and avoid excessive locking [1/5] high-low watermar
Posted by Balbir Singh on Thu, 14 Feb 2008 08:48:33 GMT
View Forum Message <> Reply to Message

KAMEZAWA Hiroyuki wrote:
> Add High/Low watermark support to res_counter.
>
> This adds high/low watermark support to res_counter.
>
> Adds new member hwmark, lwmark, wmark_state to res_counter struct.
>
> Users of res_counter must keep lwmark <= hwmark <= limit.
> (set params as lwmark == hwmark == limit will disable this feature.)

>
> wmark_state is added to read wmark status without lock.
>
> Codes will be like this. (check status after charge.)
>
> if (res_counter_charge(cnt...)) { <----(need lock)
>   .....
> }
> if (res_counter_above_hwmark(cnt)) <---- (no lock)
>   trigger background jobs...
>
> If I have to check under lock, please teach me.
>

If there are several processes running in parallel in the same cgroup, the end
result might not be so nice, specially if the usage is close to the watermarks.
I suspect that we should be OK for now, but might be worth keeping in mind.

> counter->hwmark and counter->lwmark is not automatically adjusted
> when limit is changed. So, users must change lwmark, hwmark before
> changing limit.
>
> Changelog
>   * made variable names shorter.
>   * adjusted to 2.6.24-mm1
>
> Signed-off-by: KAMEZAWA Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
>
> include/linux/res_counter.h | 30 ++++++++++++++++++++++++++
> kernel/res_counter.c | 53 ++-----
> 2 files changed, 78 insertions(+), 5 deletions(-)
>
> Index: linux-2.6.24-mm1/include/linux/res_counter.h
> ===
> --- linux-2.6.24-mm1.orig/include/linux/res_counter.h
> +++ linux-2.6.24-mm1/include/linux/res_counter.h
> @@ -19,6 +19,16 @@
>   * the helpers described beyond
>   */
>
> +/*
> + * Watermark status.
> + */
> +
> +enum watermark_state {
> + RES_WMARK_BELOW_LOW = 0, /* usage < low <= high <= max */
> + RES_WMARK_ABOVE_LOW, /* low <= usage < high <= max */

> + RES_WMARK_ABOVE_HIGH, /* low <= high <= usage <= max */
> +};
> +
> struct res_counter {
>   /*
>   * the current resource consumption level
> @@ -33,10 +43,18 @@ struct res_counter {
>   */
>   unsigned long long failcnt;
>   /*
> + * for supporting High/Low watermak
> + * Must keep low <= high <= limit.
> + */
> + unsigned long long hwmark;
> + unsigned long long lwmark;
> + enum watermark_state wmark_state; /* changed at charge/uncharge */
> + /*
>   * the lock to protect all of the above.
>   * the routines below consider this to be IRQ-safe
>   */
>   spinlock_t lock;
> +
> };
>
> /*
> @@ -66,6 +84,8 @@ enum {
>   RES_USAGE,
>   RES_LIMIT,
>   RES_FAILCNT,
> + RES_HWMARK,
> + RES_LWMARK,
> };
>
> /*
> @@ -124,4 +144,14 @@ static inline bool res_counter_check_und
>   return ret;
> }
>
> +static inline bool res_counter_below_lwmark(struct res_counter *cnt)
> +{
> + smp_rmb();
> + return (cnt->wmark_state == RES_WMARK_BELOW_LOW);
> +}
> +static inline bool res_counter_above_hwmark(struct res_counter *cnt)
> +{
> + smp_rmb();
> + return (cnt->wmark_state == RES_WMARK_ABOVE_HIGH);
> +}

> #endif
> Index: linux-2.6.24-mm1/kernel/res_counter.c
> ===
> --- linux-2.6.24-mm1.orig/kernel/res_counter.c
> +++ linux-2.6.24-mm1/kernel/res_counter.c
> @@ -17,16 +17,29 @@ void res_counter_init(struct res_counter
> {
>   spin_lock_init(&counter->lock);
>   counter->limit = (unsigned long long)LLONG_MAX;
> + counter->lwmark = (unsigned long long)LLONG_MAX;
> + counter->hwmark = (unsigned long long)LLONG_MAX;
> + counter->wmark_state = RES_WMARK_BELOW_LOW;
> }
>
> int res_counter_charge_locked(struct res_counter *counter, unsigned long val)
> {
> - if (counter->usage + val > counter->limit) {
> + unsigned long long newval = counter->usage + val;
> + if (newval > counter->limit) {
>   counter->failcnt++;
>   return -ENOMEM;
>   }
>
> - counter->usage += val;
> +   if (newval > counter->hwmark) {
> + counter->wmark_state = RES_WMARK_ABOVE_HIGH;
> + smp_wmb();

Do we need a barrier here? I suspect not, could you please document as to why a
barrier is needed?

> + } else if (newval > counter->lwmark) {
> + counter->wmark_state = RES_WMARK_ABOVE_LOW;
> + smp_wmb();

>> Ditto

> + }
> +
> + counter->usage = newval;
> +
>   return 0;
> }
>
> @@ -43,10 +56,18 @@ int res_counter_charge(struct res_counte
>
> void res_counter_uncharge_locked(struct res_counter *counter, unsigned long val)
> {

```
> + unsigned long long newval = counter->usage - val;
>   if (WARN_ON(counter->usage < val))
> -  val = counter->usage;
> +  newval = 0;
>
> - counter->usage -= val;
> + if (newval < counter->lwmark) {
> +  counter->wmark_state = RES_WMARK_BELOW_LOW;
> +  smp_wmb();

>> Ditto

> + } else if (newval < counter->hwmark) {
> +  counter->wmark_state = RES_WMARK_ABOVE_LOW;
> +  smp_wmb();

>> Ditto

> + }
> + counter->usage = newval;
>  }
>
>  void res_counter_uncharge(struct res_counter *counter, unsigned long val)
> @@ -69,6 +90,10 @@ 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();
> @@ -123,10 +148,28 @@ ssize_t res_counter_write(struct res_cou
>     goto out_free;
>   }
>   spin_lock_irqsave(&counter->lock, flags);
> + switch (member) {
> +  case RES_LIMIT:
> +   if (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:
>

--
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] memory controller : backgorund reclaim and avoid excessive locking [2/5] background reclai
Posted by KAMEZAWA Hiroyuki on Thu, 14 Feb 2008 08:57:03 GMT
View Forum Message <> Reply to Message

On Thu, 14 Feb 2008 14:02:44 +0530
Balbir Singh <balbir@linux.vnet.ibm.com> wrote:

> KAMEZAWA Hiroyuki wrote:
> > A patch for background reclaim based on high-low watermak in res_counter.
> > The daemon is called as "memcontd", here.
> >
> > Implements following:
> > * If res->usage is higher than res->hwmark, start memcontd.
> > * memcontd calls try_to_free_pages.
> > * memcontd stops if res->usage is lower than res->lwmark.
> >
>

> Can we call the dameon memcgroupd?
>
ok.

> >
> > + if (res_counter_above_hwmark(&mem->res))
> > + mem_cgroup_schedule_daemon(mem);
> > +
>
> I don't understand this part, some comments might be good here. Why do we wake
> up the dameon in mem_cgroup_isolate_pages? Why not do so under memory pressure
> like the soft limit patches does? I suspect that these patches and soft limit
> patches might have some overlap.
>
Ah, diff option is bad. this is in mem_cgroup_charge_common().
Sorry for confusion.

Final image of this series is.
==

```
	while (mem_cgroup_borrow_and_charge(mem, PAGE_SIZE)) {
		int ret;
		if (!(gfp_mask & __GFP_WAIT))
			goto out;

		if (((gfp_mask & (__GFP_FS|__GFP_IO)) != (__GFP_FS|__GFP_IO))
		    || mem_cgroup_throttle_reclaim(mem)) {
			ret = try_to_free_mem_cgroup_pages(mem, gfp_mask);
			atomic_dec(&mem->throttle.reclaimers);
			if (waitqueue_active(&mem->throttle.waitq))
				wake_up_all(&mem->throttle.waitq);
			if (ret)
				continue;
		} else {
			mem_cgroup_wait_reclaim(mem);
			continue;
		}

		/*
		 * 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;
```

```
                if (nr_retries < MEM_CGROUP_RECLAIM_RETRIES)
                        drain_all_borrow(mem);

                if (!nr_retries--) {
                        mem_cgroup_out_of_memory(mem, gfp_mask);
                        goto out;
                }
                congestion_wait(WRITE, HZ/10);
        }

        if (res_counter_above_hwmark(&mem->res))
                mem_cgroup_schedule_daemon(mem);
==


> > +  finish_wait(&mem->daemon.waitq, &wait);
> > +  try_to_free_mem_cgroup_pages(mem, GFP_HIGHUSER_MOVABLE);
> > +  /* Am I in hurry ? */
> > +  if (!res_counter_above_hwmark(&mem->res)) {
> > +   /*
> > +    * Extra relaxing..memory reclaim is hevay work.
> > +    * we don't know there is I/O congestion or not.
> > +    * So use just relax rather than congesiton_wait().
> > +    * HZ/10 is widely used value under /mm.
> > +    */
> > +   schedule_timeout(HZ/10);
>
> Why not just yeild() here and let another iteration push us back to our low
> water mark?
>
just used timeout value which is widely used under /mm.
Hmm, I'll put cond_resched() or yield() here.
And see how cpu is used.


> >
> > +static DEFINE_MUTEX(modify_param_mutex);
> >  static ssize_t mem_cgroup_write(struct cgroup *cont, struct cftype *cft,
> >      struct file *file, const char __user *userbuf,
> >      size_t nbytes, loff_t *ppos)
> >  {
> > - return res_counter_write(&mem_cgroup_from_cont(cont)->res,
> > -    cft->private, userbuf, nbytes, ppos,
> > + int ret;
> > + struct mem_cgroup *mem = mem_cgroup_from_cont(cont);
> > +
> > + mutex_lock(&modify_param_mutex);
> > + /* Attach new background reclaim daemon.
```

> > +   This must be done before change values (for easy error handling */
> > +
> > + if (cft->private == RES_HWMARK &&
> > +   !mem->daemon.kthread) {
> > + struct task_struct *thr;
> > + thr = kthread_run(mem_cgroup_reclaim_daemon, mem, "memcontd");
> > + if (IS_ERR(thr)) {
> > +   ret = PTR_ERR(thr);
> > +   goto out;
> > + }
> > + mem->daemon.kthread = thr;
> > + }
> > + ret = res_counter_write(&mem->res, cft->private, userbuf, nbytes, ppos,
> >   mem_cgroup_write_strategy);
> > +
> > + /* Even on error, don't stop reclaim daemon here. not so problematic. */
> > +
> > +out:
> > + mutex_unlock(&modify_param_mutex);
> > + return ret;
> > }
> >
>
> Could we document the changes to this function. It looks like writing a value to
> the watermarks can cause the dameon to be scheduled.
ok.
(*) creating daemon at initilization is not simple because root cgroup initilization
    is very fast...

Thank you.
-Kame

Subject: Re: [RFC] memory controller : backgorund reclaim and avoid excessive locking [1/5] high-low watermar
Posted by KAMEZAWA Hiroyuki on Thu, 14 Feb 2008 09:10:56 GMT
View Forum Message <> Reply to Message

On Thu, 14 Feb 2008 14:18:33 +0530
Balbir Singh <balbir@linux.vnet.ibm.com> wrote:
> > If I have to check under lock, please teach me.
> >
>

> If there are several processes running in parallel in the same cgroup, the end
> result might not be so nice, specially if the usage is close to the watermarks.
> I suspect that we should be OK for now, but might be worth keeping in mind.
>
I'll add text somewhere.

> > - counter->usage += val;
> > + if (newval > counter->hwmark) {
> > + counter->wmark_state = RES_WMARK_ABOVE_HIGH;
> > + smp_wmb();
>
> Do we need a barrier here? I suspect not, could you please document as to why a
> barrier is needed?
>

just chainging value with smp_wmb() and read value after smp_rmb().
By this, I think we can expect we can read snapshot value of wmark_state at
smp_rmb().
.....I misunderstand that spin_unlock() has no barrier().
ok, I'll remove smp_wmb() here.

Thanks,
-Kame

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

---

Subject: Re: [RFC] memory controller : backgorund reclaim and avoid excessive locking [3/5] throttling
Posted by Balbir Singh on Thu, 14 Feb 2008 09:14:50 GMT
View Forum Message <> Reply to Message

KAMEZAWA Hiroyuki wrote:
> Throttle memory reclaim interface for cgroup.
>
> This patch adds..
> - a limit for simultaneous callers of try_to_free_mem_cgroup_pages().
> - interface for that. memory.shrinkers.
>
> There are some reasons.
> - try_to_free... is very heavy and shoulnd't be called too much at once.
> - When the number of callers of try_to_free.. is big, we'll reclaim
>   too much memory.

>
> By this interface, a user can control the # of threads which can enter
> try_to_free...
>

What happens to the other threads, do they sleep?

> Default is 10240 ...a enough big number for unlimited. Maybe this should
> be changed.
>
>
> Changes from previous one.
>   - Added an interface to control the limit.
>   - don't call wake_up at uncharge()...it seems hevay..
>   Instead of that, sleepers use schedule_timeout(HZ/100). (10ms)
>
> Considerations:
>   - Should we add this 'throttle' to global_lru at first ?

Hmm.. interesting question. I think Rik is looking into some of these issues

>   - Do we need more knobs ?
>   - Should default value to be automtically estimated value ?
>   (I used '# of cpus/4' as default in previous version.)
>
>
> Signed-off-by: KAMEZAWA Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
>
>
> mm/memcontrol.c | 235 ++++++++++++++++++++++++++++++--------------------
> 1 files changed, 127 insertions(+), 108 deletions(-)
>
> Index: linux-2.6.24-mm1/mm/memcontrol.c
> ===
> --- linux-2.6.24-mm1.orig/mm/memcontrol.c
> +++ linux-2.6.24-mm1/mm/memcontrol.c
> @@ -145,8 +145,17 @@ struct mem_cgroup {
>   wait_queue_head_t waitq;
>   struct task_struct *kthread;
>   } daemon;
> + /*
> + * throttling params for reclaim.
> + */
> + struct {
> + int limit;

We might want to use something else instead of limit. How about
max_parallel_reclaimers?

> + atomic_t reclaimers;
> + wait_queue_head_t waitq;
> + } throttle;
> };
>
> +
> /*
>   * We use the lower bit of the page->page_cgroup pointer as a bit spin
>   * lock. We need to ensure that page->page_cgroup is atleast two
> @@ -520,6 +529,27 @@ static inline void mem_cgroup_schedule_d
>   wake_up_interruptible(&mem->daemon.waitq);
> }
>
> +static inline void mem_cgroup_wait_reclaim(struct mem_cgroup *mem)
> +{
> + DEFINE_WAIT(wait);
> + while (1) {
> + prepare_to_wait(&mem->throttle.waitq, &wait,
> +   TASK_INTERRUPTIBLE);
> + if (res_counter_check_under_limit(&mem->res)) {
> +   finish_wait(&mem->throttle.waitq, &wait);
> +   break;
> + }
> + /* expect some progress in... */
> + schedule_timeout(HZ/50);

Can't we wait on someone to wake us up? Why do we need to do a schedule_timeout? And why HZ/50 and not HZ/10? Too many timers distract the system from power management :)

> + finish_wait(&mem->throttle.waitq, &wait);
> + }
> +}
> +
> +static inline int mem_cgroup_throttle_reclaim(struct mem_cgroup *mem)
> +{
> + return atomic_add_unless(&mem->throttle.reclaimers, 1,
> +   mem->throttle.limit);
> +}
>
> unsigned long mem_cgroup_isolate_pages(unsigned long nr_to_scan,
>   struct list_head *dst,
> @@ -652,11 +682,22 @@ retry:
>   * the cgroup limit.
>   */
>   while (res_counter_charge(&mem->res, PAGE_SIZE)) {
> + int ret;

>   if (!(gfp_mask & __GFP_WAIT))
>   goto out;
>
> - if (try_to_free_mem_cgroup_pages(mem, gfp_mask))
> + if (((gfp_mask & (__GFP_FS|__GFP_IO)) != (__GFP_FS|__GFP_IO))
> + || mem_cgroup_throttle_reclaim(mem)) {

I think we should split this check and add detailed comments. If we cannot
sleep, then we cannot be throttled, right?

> +   ret = try_to_free_mem_cgroup_pages(mem, gfp_mask);
> +   atomic_dec(&mem->throttle.reclaimers);
> +   if (waitqueue_active(&mem->throttle.waitq))
> +   wake_up_all(&mem->throttle.waitq);
> +   if (ret)
> +   continue;
> + } else {
> +   mem_cgroup_wait_reclaim(mem);
>   continue;
> + }
>
>   /*
>   * try_to_free_mem_cgroup_pages() might not give us a full
> @@ -1054,6 +1095,19 @@ static ssize_t mem_force_empty_read(stru
>   return -EINVAL;
>   }
>
> +static int mem_throttle_write(struct cgroup *cont, struct cftype *cft, u64 val)
> +{
> + struct mem_cgroup *mem = mem_cgroup_from_cont(cont);
> + int limit = (int)val;
> + mem->throttle.limit = limit;
> + return 0;
> +}
> +
> +static u64 mem_throttle_read(struct cgroup *cont, struct cftype *cft)
> +{
> + struct mem_cgroup *mem = mem_cgroup_from_cont(cont);
> + return (u64)mem->throttle.limit;
> +}
>
>   static const struct mem_cgroup_stat_desc {
>   const char *msg;
> @@ -1146,6 +1200,11 @@ static struct cftype mem_cgroup_files[]
>   .read = mem_force_empty_read,
>   },
>   {
> + .name = "shrinks",

> + .write_uint = mem_throttle_write,
> + .read_uint = mem_throttle_read,
> + },
> + {
>   .name = "stat",
>   .open = mem_control_stat_open,
>   },
> @@ -1215,6 +1274,11 @@ mem_cgroup_create(struct cgroup_subsys *
>   goto free_out;
>   init_waitqueue_head(&mem->daemon.waitq);
>   mem->daemon.kthread = NULL;
> +
> + init_waitqueue_head(&mem->throttle.waitq);
> + mem->throttle.limit = 10240; /* maybe enough big for no throttle */

Why 10240? So, 10,000 threads can run in parallel, isn't that an overkill? How
about setting it to number of cpus/2 or something like that?

> + atomic_set(&mem->throttle.reclaimers, 0);
> +
>   return &mem->css;
> free_out:
>   for_each_node_state(node, N_POSSIBLE)
>

--
 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] memory controller : backgorund reclaim and avoid excessive locking [3/5] throttling
Posted by KAMEZAWA Hiroyuki on Thu, 14 Feb 2008 09:38:56 GMT
View Forum Message <> Reply to Message

On Thu, 14 Feb 2008 14:44:50 +0530
Balbir Singh <balbir@linux.vnet.ibm.com> wrote:
> > By this interface, a user can control the # of threads which can enter
> > try_to_free...
> >
>

> What happens to the other threads, do they sleep?
>
yes.

> > Default is 10240 ...a enough big number for unlimited. Maybe this should
> > be changed.
> >
> >
> > Changes from previous one.
> > - Added an interface to control the limit.
> > - don't call wake_up at uncharge()...it seems hevay..
> >   Instead of that, sleepers use schedule_timeout(HZ/100). (10ms)
> >
> > Considerations:
> > - Should we add this 'throttle' to global_lru at first ?
>
> Hmm.. interesting question. I think Rik is looking into some of these issues
>
Maybe. I thought someone posted a patch to -mm but I lost the mail.

> > + * throttling params for reclaim.
> > + */
> > + struct {
> > + int limit;
>
> We might want to use something else instead of limit. How about
> max_parallel_reclaimers?
>
ok.

> > + prepare_to_wait(&mem->throttle.waitq, &wait,
> > +   TASK_INTERRUPTIBLE);
> > + if (res_counter_check_under_limit(&mem->res)) {
> > +   finish_wait(&mem->throttle.waitq, &wait);
> > +   break;
> > + }
> > + /* expect some progress in... */
> > + schedule_timeout(HZ/50);
>
> Can't we wait on someone to wake us up? Why do we need to do a schedule_timeout?

A method "Someone wakes up" adds new code to uncharge.

 - check value and limt.
 - check waiters
 - wakes them up if any (and touches other cpu's runq->lock.)

Not very light-weight ops. just polling works well in this case, I think.

> And why HZ/50 and not HZ/10? Too many timers distract the system from power
> management :)
ok, HZ/10. (or add some knob.)

> > - if (try_to_free_mem_cgroup_pages(mem, gfp_mask))
> > + if (((gfp_mask & (__GFP_FS|__GFP_IO)) != (__GFP_FS|__GFP_IO))
> > + || mem_cgroup_throttle_reclaim(mem)) {
>
> I think we should split this check and add detailed comments. If we cannot
> sleep, then we cannot be throttled, right?
>
ok.

> > + init_waitqueue_head(&mem->throttle.waitq);
> > + mem->throttle.limit = 10240; /* maybe enough big for no throttle */
>
> Why 10240? So, 10,000 threads can run in parallel, isn't that an overkill?

yes, it's overkill. Because I don't want to define *unlimted* value,
I used a big number.

> How about setting it to number of cpus/2 or something like that?

Hmm, in previous version, cpus/4 was used as default.
some value like cpus/2, nodes/2, seems good. (maybe).

A concern is that memory cgroup can be used with cpuset or
some other controllers. So, the best number can be various under environments.

It's one of choice we set the value as unlimited and makes user set suitable by hand.

Thanks,
-Kame

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

---

Subject: Re: [RFC] memory controller : backgorund reclaim and avoid excessive

> + /* try to charge */
> + ret = res_counter_charge(&mem->res, mem->borrow_unit);
> + if (!ret) { /* success */
> + *bwp += (mem->borrow_unit - size);
> + goto out;
> + }
> + }
> + spin_lock(&mem->res.lock);
> + ret = res_counter_charge_locked(&mem->res, size);
> + spin_unlock(&mem->res.lock);

although i don't know if it matters, this retrying of charge affects failcnt.

> +static void mem_cgroup_return_and_uncharge(struct mem_cgroup *mem, int size)
> +{
> + unsigned long flags;
> + int uncharge_size = 0;
> +
> + local_irq_save(flags);
> + if (mem->borrow_unit) {
> + int limit = mem->borrow_unit * 2;
> + int cpu;
> + s64 *bwp;
> + cpu = smp_processor_id();
> + bwp = &mem->stat.cpustat[cpu].count[MEM_CGROUP_STAT_BORROW];
> + *bwp += size;
> + if (*bwp > limit) {
> + uncharge_size = *bwp - mem->borrow_unit;
> + *bwp = mem->borrow_unit;
> + }
> + } else
> + uncharge_size = size;
> +
> + if (uncharge_size) {
> + spin_lock(&mem->res.lock);
> + res_counter_uncharge_locked(&mem->res, size);

s/size/uncharge_size/

> @@ -1109,12 +1202,29 @@ static u64 mem_throttle_read(struct cgro
> return (u64)mem->throttle.limit;
> }
>
> +static int mem_bulkratio_write(struct cgroup *cont, struct cftype *cft, u64 val)
> +{

> + struct mem_cgroup *mem = mem_cgroup_from_cont(cont);
> + int unit = val * PAGE_SIZE;
> + if (unit > (PAGE_SIZE << (MAX_ORDER/2)))
> + return -EINVAL;
> + mem->borrow_unit = unit;
> + return 0;
> +}

it seems unsafe with concurrent mem_cgroup_borrow_and_charge or
mem_cgroup_return_and_uncharge.

YAMAMOTO Takashi

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

---

Subject: Re: [RFC] memory controller : backgorund reclaim and avoid excessive locking [5/5] lazy page_cgroup
Posted by yamamoto on Mon, 18 Feb 2008 01:58:40 GMT
View Forum Message <> Reply to Message

> + /*
> + * For lazy freeing (not GC)
> + */
> + struct {
> + struct mem_cgroup_per_zone *mz;
> + int   num;
> +#define GARBAGE_MAXSIZE (16)
> + struct page_cgroup *vec[GARBAGE_MAXSIZE];
> + } garbage[NR_CPUS];
> };

i think you want to dedicate cache lines.

> @@ -176,12 +185,14 @@ struct page_cgroup {
>   struct list_head lru; /* per cgroup LRU list */
>   struct page *page;
>   struct mem_cgroup *mem_cgroup;
> + struct mem_cgroup_per_zone *mz; /* now belongs to...*/

is this for performance?

> @@ -408,10 +427,12 @@ static void __mem_cgroup_move_lists(stru
>   if (active) {
>   MEM_CGROUP_ZSTAT(mz, MEM_CGROUP_ZSTAT_ACTIVE) += 1;
>   pc->flags |= PAGE_CGROUP_FLAG_ACTIVE;

> + pc->mz = mz;
>   list_move(&pc->lru, &mz->active_list);
>   } else {
>   MEM_CGROUP_ZSTAT(mz, MEM_CGROUP_ZSTAT_INACTIVE) += 1;
>   pc->flags &= ~PAGE_CGROUP_FLAG_ACTIVE;
> + pc->mz = mz;
>   list_move(&pc->lru, &mz->inactive_list);
>   }
> }

isn't pc->mz already assigned by __mem_cgroup_add_list?

> @@ -1050,11 +1114,15 @@ mem_cgroup_force_empty_list(struct mem_c
>   if (list_empty(list))
>   return;
> retry:
> + all_free_garbages(mem);
>   count = FORCE_UNCHARGE_BATCH;
>   spin_lock_irqsave(&mz->lru_lock, flags);
>
>   while (--count && !list_empty(list)) {
>   pc = list_entry(list->prev, struct page_cgroup, lru);
> + /* If there are still garbage, exit and retry */
> + if (pc->flags & PAGE_CGROUP_FLAG_GARBAGE)
> +   break;

i think mem_cgroup_isolate_pages needs a similar check.

YAMAMOTO Takashi

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

---

Subject: Re: [RFC] memory controller : backgorund reclaim and avoid excessive locking [4/5] borrow resource
Posted by KAMEZAWA Hiroyuki on Mon, 18 Feb 2008 02:05:53 GMT
View Forum Message <> Reply to Message

On Mon, 18 Feb 2008 09:53:35 +0900 (JST)
yamamoto@valinux.co.jp (YAMAMOTO Takashi) wrote:

> > + /* try to charge */
> > + ret = res_counter_charge(&mem->res, mem->borrow_unit);
> > + if (!ret) { /* success */
> > +   *bwp += (mem->borrow_unit - size);
> > +   goto out;

> > + }
> > + }
> > + spin_lock(&mem->res.lock);
> > + ret = res_counter_charge_locked(&mem->res, size);
> > + spin_unlock(&mem->res.lock);
>
> although i don't know if it matters, this retrying of charge affects failcnt.
>
Hmm, yes. But what failcnt means is not so clear anyway.

> > +static void mem_cgroup_return_and_uncharge(struct mem_cgroup *mem, int size)
> > +{
> > + unsigned long flags;
> > + int uncharge_size = 0;
> > +
> > + local_irq_save(flags);
> > + if (mem->borrow_unit) {
> > + int limit = mem->borrow_unit * 2;
> > + int cpu;
> > + s64 *bwp;
> > + cpu = smp_processor_id();
> > + bwp = &mem->stat.cpustat[cpu].count[MEM_CGROUP_STAT_BORROW];
> > + *bwp += size;
> > + if (*bwp > limit) {
> > + uncharge_size = *bwp - mem->borrow_unit;
> > + *bwp = mem->borrow_unit;
> > + }
> > + } else
> > + uncharge_size = size;
> > +
> > + if (uncharge_size) {
> > + spin_lock(&mem->res.lock);
> > + res_counter_uncharge_locked(&mem->res, size);
>
> s/size/uncharge_size/
>
Oops...will fix.

> > @@ -1109,12 +1202,29 @@ static u64 mem_throttle_read(struct cgro
> > return (u64)mem->throttle.limit;
> > }
> >
> > +static int mem_bulkratio_write(struct cgroup *cont, struct cftype *cft, u64 val)
> > +{
> > + struct mem_cgroup *mem = mem_cgroup_from_cont(cont);
> > + int unit = val * PAGE_SIZE;

> > + if (unit > (PAGE_SIZE << (MAX_ORDER/2)))
> > + return -EINVAL;
> > + mem->borrow_unit = unit;
> > + return 0;
> > +}
>
> it seems unsafe with concurrent mem_cgroup_borrow_and_charge or
> mem_cgroup_return_and_uncharge.
>
Hmm, making this to be not configurable will be good.

Thanks,
-Kame

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

---

Subject: Re: [RFC] memory controller : backgorund reclaim and avoid excessive locking [5/5] lazy page_cgroup
Posted by KAMEZAWA Hiroyuki on Mon, 18 Feb 2008 02:11:05 GMT
View Forum Message <> Reply to Message

On Mon, 18 Feb 2008 10:58:40 +0900 (JST)
yamamoto@valinux.co.jp (YAMAMOTO Takashi) wrote:

> > + /*
> > + * For lazy freeing (not GC)
> > + */
> > + struct {
> > + struct mem_cgroup_per_zone *mz;
> > + int   num;
> > +#define GARBAGE_MAXSIZE (16)
> > + struct page_cgroup *vec[GARBAGE_MAXSIZE];
> > + } garbage[NR_CPUS];
> > };
>
> i think you want to dedicate cache lines.
>
> > @@ -176,12 +185,14 @@ struct page_cgroup {
> >   struct list_head lru; /* per cgroup LRU list */
> >   struct page *page;
> >   struct mem_cgroup *mem_cgroup;
> > + struct mem_cgroup_per_zone *mz; /* now belongs to...*/
>
> is this for performance?

>
We need to find what zone pc is under...
Now, we can find it by pc->page.
But I don't want to trust pc->page of freed pages.

> > @@ -408,10 +427,12 @@ static void __mem_cgroup_move_lists(stru
> > if (active) {
> > MEM_CGROUP_ZSTAT(mz, MEM_CGROUP_ZSTAT_ACTIVE) += 1;
> > pc->flags |= PAGE_CGROUP_FLAG_ACTIVE;
> > + pc->mz = mz;
> > list_move(&pc->lru, &mz->active_list);
> > } else {
> > MEM_CGROUP_ZSTAT(mz, MEM_CGROUP_ZSTAT_INACTIVE) += 1;
> > pc->flags &= ~PAGE_CGROUP_FLAG_ACTIVE;
> > + pc->mz = mz;
> > list_move(&pc->lru, &mz->inactive_list);
> > }
> > }
>
> isn't pc->mz already assigned by __mem_cgroup_add_list?
>
Ah, yes. I can remove this.

> > @@ -1050,11 +1114,15 @@ mem_cgroup_force_empty_list(struct mem_c
> > if (list_empty(list))
> > return;
> > retry:
> > + all_free_garbages(mem);
> > count = FORCE_UNCHARGE_BATCH;
> > spin_lock_irqsave(&mz->lru_lock, flags);
> >
> > while (--count && !list_empty(list)) {
> > pc = list_entry(list->prev, struct page_cgroup, lru);
> > + /* If there are still garbage, exit and retry */
> > + if (pc->flags & PAGE_CGROUP_FLAG_GARBAGE)
> > + break;
>
> i think mem_cgroup_isolate_pages needs a similar check.
>
Yes. maybe my refresh miss..

Thanks,
-Kame

> YAMAMOTO Takashi
>

Subject: Re: [RFC] memory controller : backgorund reclaim and avoid excessive locking [5/5] lazy page_cgroup
Posted by Balbir Singh on Mon, 18 Feb 2008 04:35:16 GMT
View Forum Message <> Reply to Message

KAMEZAWA Hiroyuki wrote:
> A big lock contetion of memory controller is mz->lru_lock.
>
> This is acquired when
> 1. add to lru
> 2. remove from lru
> 3. scan lru list
>
> It seems 1. and 3. are unavoidable. but 2. can be delayed.
>
> This patch make removing page_cgorup from lru-list be lazy and batched.
> (Like pagevec..)
>
> This patch adds a new flag page_cgroup and make lru scan routine
> ignores it.
>
>
> I think this reduces lock contention especially when
> - several tasks are exiting at once.
> - several files are removed at once.
>
>
> Signed-off-by: KAMEZAWA Hiroyuki <kamezawa.hiroyu@jp.fujitsu.com>
>

Hi, KAMEZAWA-San,

For this and the next patch, do you know if there is a performance improvement? Have we measured it. The complexity of the code seems to be high, want to make sure it is worth it.

Could we re-use the pagevec mechansim for LRU handling? That was my long term plan and I also wanted to move the kmalloc's to their own cache and use kmem_cache_alloc and try some other experiments. May be batching the alloc's for page_container can be done in the same way that you've proposed LRU and accounting changes.

--
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] memory controller : backgorund reclaim and avoid excessive locking [5/5] lazy page_cgroup
Posted by KAMEZAWA Hiroyuki on Mon, 18 Feb 2008 04:54:03 GMT
View Forum Message <> Reply to Message

On Mon, 18 Feb 2008 10:05:16 +0530
Balbir Singh <balbir@linux.vnet.ibm.com> wrote:
> Hi, KAMEZAWA-San,
>
> For this and the next patch, do you know if there is a performance improvement?
> Have we measured it. The complexity of the code seems to be high, want to make
> sure it is worth it.

I'd like to make codes simpler (by removing knobs and...)
I confirmed lock-contention/cache bouncing was much reduced.
(you can see it by /proc/lock_stat)
But I have no number now. I'm now searching an apps which can show performance
improvement clearly and wait for avilable machine ;)
So, please wait. I'm not hurrying.

> Could we re-use the pagevec mechansim for LRU handling?
Hmm, it's also complicated.

>That was my long term plan and I also wanted to move the kmalloc's to their
>own cache and use kmem_cache_alloc and try some other experiments.
maybe good idea.

> May be batching the alloc's for
> page_container can be done in the same way that you've proposed LRU and
> accounting changes.
>
Maybe.

Thanks,
-Kame

_____