
Subject: [RFC] [PATCH] memory controller background reclamation

Posted by [yamamoto](#) on Mon, 22 Oct 2007 09:43:49 GMT

[View Forum Message](#) <> [Reply to Message](#)

hi,

i implemented background reclamation for your memory controller and did a few benchmarks with and without it. any comments?

YAMAMOTO Takashi

"time make -j4 bzImage" in a cgroup with 64MB limit:

without patch:

```
real 22m22.389s
user 13m35.163s
sys 6m12.283s
```

memory.failcnt after a run: 106647

with patch:

```
real 19m25.860s
user 14m10.549s
sys 6m13.831s
```

memory.failcnt after a run: 35366

"for x in 1 2 3;do time dd if=/dev/zero bs=1M count=300|tail;done"
in a cgroup with 16MB limit:

without patch:

```
300+0 records in
300+0 records out
314572800 bytes (315 MB) copied, 19.0058 seconds, 16.6 MB/s
u 0.00s s 0.67s e 19.01s maj 90 min 2021 in 0 out 0
u 0.48s s 7.22s e 93.13s maj 20475 min 210903 in 0 out 0
300+0 records in
300+0 records out
314572800 bytes (315 MB) copied, 14.8394 seconds, 21.2 MB/s
u 0.00s s 0.63s e 14.84s maj 53 min 1392 in 0 out 0
u 0.48s s 7.00s e 94.39s maj 20125 min 211145 in 0 out 0
300+0 records in
300+0 records out
314572800 bytes (315 MB) copied, 14.0635 seconds, 22.4 MB/s
u 0.01s s 0.59s e 14.07s maj 50 min 1385 in 0 out 0
u 0.57s s 6.93s e 91.54s maj 20359 min 210911 in 0 out 0
```

memory.failcnt after a run: 8804

with patch:

```
300+0 records in
300+0 records out
314572800 bytes (315 MB) copied, 5.94875 seconds, 52.9 MB/s
u 0.00s s 0.67s e 5.95s maj 206 min 5393 in 0 out 0
u 0.45s s 6.63s e 46.07s maj 21350 min 210252 in 0 out 0
300+0 records in
300+0 records out
314572800 bytes (315 MB) copied, 8.16441 seconds, 38.5 MB/s
u 0.00s s 0.60s e 8.17s maj 240 min 4614 in 0 out 0
u 0.45s s 6.56s e 54.56s maj 22298 min 209255 in 0 out 0
300+0 records in
300+0 records out
314572800 bytes (315 MB) copied, 4.60967 seconds, 68.2 MB/s
u 0.00s s 0.60s e 4.64s maj 196 min 4733 in 0 out 0
u 0.46s s 6.53s e 49.28s maj 22223 min 209384 in 0 out 0
```

memory.failcnt after a run: 1589

```
--- linux-2.6.23-rc8-mm2-cgkswapd/mm/memcontrol.c.BACKUP 2007-10-01 17:19:57.000000000
+0900
+++ linux-2.6.23-rc8-mm2-cgkswapd/mm/memcontrol.c 2007-10-22 13:46:01.000000000 +0900
@@ -27,6 +27,7 @@
#include <linux/rcupdate.h>
#include <linux/swap.h>
#include <linux/spinlock.h>
+#include <linux/workqueue.h>
#include <linux/fs.h>

#include <asm/uaccess.h>
@@ -63,6 +64,8 @@ struct mem_cgroup {
    */
    spinlock_t lru_lock;
    unsigned long control_type; /* control RSS or RSS+Pagecache */
+
+ struct work_struct reclaim_work;
};

/*
@@ -95,6 +98,9 @@ enum {

static struct mem_cgroup init_mem_cgroup;
```

```

+static DEFINE_MUTEX(mem_cgroup_workqueue_init_lock);
+static struct workqueue_struct *mem_cgroup_workqueue;
+
+static inline
+struct mem_cgroup *mem_cgroup_from_cont(struct cgroup *cont)
+{
@@ -250,6 +256,69 @@ unsigned long mem_cgroup_isolate_pages(u
+return nr_taken;
+}

+static int
+mem_cgroup_need_reclaim(struct mem_cgroup *mem)
+{
+struct res_counter *cnt = &mem->res;
+int doreclaim;
+unsigned long flags;
+
+/* XXX should be in res_counter */
+/* XXX should not hardcode a watermark */
+spin_lock_irqsave(&cnt->lock, flags);
+doreclaim = cnt->usage > cnt->limit / 4 * 3;
+spin_unlock_irqrestore(&cnt->lock, flags);
+
+return doreclaim;
+}
+
+static void
+mem_cgroup_schedule_reclaim_if_necessary(struct mem_cgroup *mem)
+{
+
+if (mem_cgroup_workqueue == NULL) {
+BUG_ON(mem->css.cgroup->parent != NULL);
+return;
+}
+
+if (work_pending(&mem->reclaim_work))
+return;
+
+if (!mem_cgroup_need_reclaim(mem))
+return;
+
+css_get(&mem->css); /* XXX need some thoughts wrt cgroup removal. */
+/*
+ * XXX workqueue is not an ideal mechanism for our purpose.
+ * revisit later.
+ */
+if (!queue_work(mem_cgroup_workqueue, &mem->reclaim_work))
+css_put(&mem->css);

```

```

+}
+
+static void
+mem_cgroup_reclaim(struct work_struct *work)
+{
+ struct mem_cgroup * const mem =
+ container_of(work, struct mem_cgroup, reclaim_work);
+ int batch_count = 128; /* XXX arbitrary */
+
+ for (; batch_count > 0; batch_count--) {
+ if (!mem_cgroup_need_reclaim(mem))
+ break;
+ /*
+ * XXX try_to_free_foo is not a correct mechanism to
+ * use here. eg. ALLOCSTALL counter
+ * revisit later.
+ */
+ if (!try_to_free_mem_cgroup_pages(mem, GFP_KERNEL))
+ break;
+ }
+ if (batch_count == 0)
+ mem_cgroup_schedule_reclaim_if_necessary(mem);
+ css_put(&mem->css);
+}
+
+/*
+ * Charge the memory controller for page usage.
+ * Return
@@ -311,6 +380,9 @@ int mem_cgroup_charge(struct page *page,
+ */
+ while (res_counter_charge(&mem->res, PAGE_SIZE)) {
+ bool is_atomic = gfp_mask & GFP_ATOMIC;
+
+ mem_cgroup_schedule_reclaim_if_necessary(mem);
+
+ /*
+ * We cannot reclaim under GFP_ATOMIC, fail the charge
+ */
@@ -551,8 +623,18 @@ mem_cgroup_create(struct cgroup_subsys *
+ if (unlikely((cont->parent) == NULL)) {
+ mem = &init_mem_cgroup;
+ init_mm.mem_cgroup = mem;
+ } else
+ } else {
+ /* XXX too late for the top-level cgroup */
+ if (mem_cgroup_workqueue == NULL) {
+ mutex_lock(&mem_cgroup_workqueue_init_lock);
+ if (mem_cgroup_workqueue == NULL) {

```

```

+ mem_cgroup_workqueue =
+   create_workqueue("mem_cgroup");
+ }
+ mutex_unlock(&mem_cgroup_workqueue_init_lock);
+ }
+ mem = kzalloc(sizeof(struct mem_cgroup), GFP_KERNEL);
+ }

if (mem == NULL)
    return NULL;
@@ -562,6 +644,7 @@ mem_cgroup_create(struct cgroup_subsys *
    INIT_LIST_HEAD(&mem->inactive_list);
    spin_lock_init(&mem->lru_lock);
    mem->control_type = MEM_CGROUP_TYPE_ALL;
+ INIT_WORK(&mem->reclaim_work, mem_cgroup_reclaim);
    return &mem->css;
}

```

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

Subject: Re: [RFC] [PATCH] memory controller background reclamation
Posted by [Balbir Singh](#) on Mon, 22 Oct 2007 16:10:45 GMT
[View Forum Message](#) <> [Reply to Message](#)

YAMAMOTO Takashi wrote:

```

> hi,
>
> i implemented background reclamation for your memory controller and
> did a few benchmarks with and without it. any comments?
>
> YAMAMOTO Takashi
>
>
> -----
> "time make -j4 bzImage" in a cgroup with 64MB limit:
>
> without patch:
> real 22m22.389s
> user 13m35.163s
> sys 6m12.283s
>
> memory.failcnt after a run: 106647
>
> with patch:

```

```

> real 19m25.860s
> user 14m10.549s
> sys 6m13.831s
>
> memory.failcnt after a run: 35366
>
> "for x in 1 2 3;do time dd if=/dev/zero bs=1M count=300|tail;done"
> in a cgroup with 16MB limit:
>
> without patch:
> 300+0 records in
> 300+0 records out
> 314572800 bytes (315 MB) copied, 19.0058 seconds, 16.6 MB/s
> u 0.00s s 0.67s e 19.01s maj 90 min 2021 in 0 out 0
> u 0.48s s 7.22s e 93.13s maj 20475 min 210903 in 0 out 0
> 300+0 records in
> 300+0 records out
> 314572800 bytes (315 MB) copied, 14.8394 seconds, 21.2 MB/s
> u 0.00s s 0.63s e 14.84s maj 53 min 1392 in 0 out 0
> u 0.48s s 7.00s e 94.39s maj 20125 min 211145 in 0 out 0
> 300+0 records in
> 300+0 records out
> 314572800 bytes (315 MB) copied, 14.0635 seconds, 22.4 MB/s
> u 0.01s s 0.59s e 14.07s maj 50 min 1385 in 0 out 0
> u 0.57s s 6.93s e 91.54s maj 20359 min 210911 in 0 out 0
>
> memory.failcnt after a run: 8804
>
> with patch:
> 300+0 records in
> 300+0 records out
> 314572800 bytes (315 MB) copied, 5.94875 seconds, 52.9 MB/s
> u 0.00s s 0.67s e 5.95s maj 206 min 5393 in 0 out 0
> u 0.45s s 6.63s e 46.07s maj 21350 min 210252 in 0 out 0
> 300+0 records in
> 300+0 records out
> 314572800 bytes (315 MB) copied, 8.16441 seconds, 38.5 MB/s
> u 0.00s s 0.60s e 8.17s maj 240 min 4614 in 0 out 0
> u 0.45s s 6.56s e 54.56s maj 22298 min 209255 in 0 out 0
> 300+0 records in
> 300+0 records out
> 314572800 bytes (315 MB) copied, 4.60967 seconds, 68.2 MB/s
> u 0.00s s 0.60s e 4.64s maj 196 min 4733 in 0 out 0
> u 0.46s s 6.53s e 49.28s maj 22223 min 209384 in 0 out 0
>
> memory.failcnt after a run: 1589

```

Looks quite nice!

```

> -----
>
>
> --- linux-2.6.23-rc8-mm2-cgkswapd/mm/memcontrol.c.BACKUP 2007-10-01
17:19:57.000000000 +0900
> +++ linux-2.6.23-rc8-mm2-cgkswapd/mm/memcontrol.c 2007-10-22 13:46:01.000000000 +0900
> @@ -27,6 +27,7 @@
> #include <linux/rcupdate.h>
> #include <linux/swap.h>
> #include <linux/spinlock.h>
> +#include <linux/workqueue.h>
> #include <linux/fs.h>
>
> #include <asm/uaccess.h>
> @@ -63,6 +64,8 @@ struct mem_cgroup {
>  */
>  spinlock_t lru_lock;
>  unsigned long control_type; /* control RSS or RSS+Pagecache */
> +
> + struct work_struct reclaim_work;
> };
>
> /*
> @@ -95,6 +98,9 @@ enum {
>
> static struct mem_cgroup init_mem_cgroup;
>
> +static DEFINE_MUTEX(mem_cgroup_workqueue_init_lock);
> +static struct workqueue_struct *mem_cgroup_workqueue;
> +
> static inline
> struct mem_cgroup *mem_cgroup_from_cont(struct cgroup *cont)
> {
> @@ -250,6 +256,69 @@ unsigned long mem_cgroup_isolate_pages(u
>  return nr_taken;
> }
>
> +static int
> +mem_cgroup_need_reclaim(struct mem_cgroup *mem)
> +{
> + struct res_counter * const cnt = &mem->res;
> + int doreclaim;
> + unsigned long flags;
> +
> + /* XXX should be in res_counter */
> + /* XXX should not hardcode a watermark */

```

We could add the following API to resource counters

```
res_counter_set_low_watermark
res_counter_set_high_watermark
res_counter_below_low_watermark
res_counter_above_high_watermark
```

and add

```
low_watermark
high_watermark
```

members to the resource group. We could push out data upto the low watermark from the cgroup.

```
> + spin_lock_irqsave(&cnt->lock, flags);
> + doreclaim = cnt->usage > cnt->limit / 4 * 3;
> + spin_unlock_irqrestore(&cnt->lock, flags);
> +
> + return doreclaim;
> +}
> +
> +static void
> +mem_cgroup_schedule_reclaim_if_necessary(struct mem_cgroup *mem)
> +{
> +
> + if (mem_cgroup_workqueue == NULL) {
> + BUG_ON(mem->css.cgroup->parent != NULL);
> + return;
> + }
> +
> + if (work_pending(&mem->reclaim_work))
> + return;
> +
> + if (!mem_cgroup_need_reclaim(mem))
> + return;
> +
> + css_get(&mem->css); /* XXX need some thoughts wrt cgroup removal. */
> + /*
> + * XXX workqueue is not an ideal mechanism for our purpose.
> + * revisit later.
> + */
> + if (!queue_work(mem_cgroup_workqueue, &mem->reclaim_work))
> + css_put(&mem->css);
> +}
> +
> +static void
> +mem_cgroup_reclaim(struct work_struct *work)
```

```

> +{
> + struct mem_cgroup * const mem =
> +     container_of(work, struct mem_cgroup, reclaim_work);
> + int batch_count = 128; /* XXX arbitrary */
> +
> + for (; batch_count > 0; batch_count--) {
> +     if (!mem_cgroup_need_reclaim(mem))
> +         break;
> +     /*
> +      * XXX try_to_free_foo is not a correct mechanism to
> +      * use here. eg. ALLOCSTALL counter
> +      * revisit later.
> +      */
> +     if (!try_to_free_mem_cgroup_pages(mem, GFP_KERNEL))

```

We could make `try_to_free_mem_cgroup_pages`, batch aware and pass that in `scan_control`.

```

> +     break;
> + }
> + if (batch_count == 0)
> +     mem_cgroup_schedule_reclaim_if_necessary(mem);
> + css_put(&mem->css);
> + }
> +
> /*
>  * Charge the memory controller for page usage.
>  * Return
> @@ -311,6 +380,9 @@ int mem_cgroup_charge(struct page *page,
>  */
> while (res_counter_charge(&mem->res, PAGE_SIZE)) {
>     bool is_atomic = gfp_mask & GFP_ATOMIC;
> +
> +     mem_cgroup_schedule_reclaim_if_necessary(mem);
> +
> /*
>  * We cannot reclaim under GFP_ATOMIC, fail the charge
>  */
> @@ -551,8 +623,18 @@ mem_cgroup_create(struct cgroup_subsys *
> if (unlikely((cont->parent) == NULL)) {
>     mem = &init_mem_cgroup;
>     init_mm.mem_cgroup = mem;
> - } else
> + } else {
> +     /* XXX too late for the top-level cgroup */
> +     if (mem_cgroup_workqueue == NULL) {
> +         mutex_lock(&mem_cgroup_workqueue_init_lock);

```

```

> + if (mem_cgroup_workqueue == NULL) {
> +     mem_cgroup_workqueue =
> +         create_workqueue("mem_cgroup");
> + }
> + mutex_unlock(&mem_cgroup_workqueue_init_lock);
> + }
> mem = kzalloc(sizeof(struct mem_cgroup), GFP_KERNEL);
> + }
>
> if (mem == NULL)
>     return NULL;
> @@ -562,6 +644,7 @@ mem_cgroup_create(struct cgroup_subsys *
> INIT_LIST_HEAD(&mem->inactive_list);
> spin_lock_init(&mem->lru_lock);
> mem->control_type = MEM_CGROUP_TYPE_ALL;
> + INIT_WORK(&mem->reclaim_work, mem_cgroup_reclaim);
> return &mem->css;
> }
>

```

--

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] [PATCH] memory controller background reclamation
 Posted by [Vaidyanathan Srinivas](#) on Mon, 22 Oct 2007 17:06:40 GMT
[View Forum Message](#) <> [Reply to Message](#)

YAMAMOTO Takashi wrote:

```

> hi,
>
> i implemented background reclamation for your memory controller and
> did a few benchmarks with and without it. any comments?
>
> YAMAMOTO Takashi
>
>
> -----
> "time make -j4 bzImage" in a cgroup with 64MB limit:
>

```

```

> without patch:
> real 22m22.389s
> user 13m35.163s
> sys 6m12.283s
>
> memory.failcnt after a run: 106647
>
> with patch:
> real 19m25.860s
> user 14m10.549s
> sys 6m13.831s
>
> memory.failcnt after a run: 35366
>
> "for x in 1 2 3;do time dd if=/dev/zero bs=1M count=300|tail;done"
> in a cgroup with 16MB limit:
>
> without patch:
> 300+0 records in
> 300+0 records out
> 314572800 bytes (315 MB) copied, 19.0058 seconds, 16.6 MB/s
> u 0.00s s 0.67s e 19.01s maj 90 min 2021 in 0 out 0
> u 0.48s s 7.22s e 93.13s maj 20475 min 210903 in 0 out 0
> 300+0 records in
> 300+0 records out
> 314572800 bytes (315 MB) copied, 14.8394 seconds, 21.2 MB/s
> u 0.00s s 0.63s e 14.84s maj 53 min 1392 in 0 out 0
> u 0.48s s 7.00s e 94.39s maj 20125 min 211145 in 0 out 0
> 300+0 records in
> 300+0 records out
> 314572800 bytes (315 MB) copied, 14.0635 seconds, 22.4 MB/s
> u 0.01s s 0.59s e 14.07s maj 50 min 1385 in 0 out 0
> u 0.57s s 6.93s e 91.54s maj 20359 min 210911 in 0 out 0
>
> memory.failcnt after a run: 8804
>
> with patch:
> 300+0 records in
> 300+0 records out
> 314572800 bytes (315 MB) copied, 5.94875 seconds, 52.9 MB/s
> u 0.00s s 0.67s e 5.95s maj 206 min 5393 in 0 out 0
> u 0.45s s 6.63s e 46.07s maj 21350 min 210252 in 0 out 0
> 300+0 records in
> 300+0 records out
> 314572800 bytes (315 MB) copied, 8.16441 seconds, 38.5 MB/s
> u 0.00s s 0.60s e 8.17s maj 240 min 4614 in 0 out 0
> u 0.45s s 6.56s e 54.56s maj 22298 min 209255 in 0 out 0
> 300+0 records in

```

```

> 300+0 records out
> 314572800 bytes (315 MB) copied, 4.60967 seconds, 68.2 MB/s
> u 0.00s s 0.60s e 4.64s maj 196 min 4733 in 0 out 0
> u 0.46s s 6.53s e 49.28s maj 22223 min 209384 in 0 out 0
>
> memory.failcnt after a run: 1589
> -----

```

Results looks good. Background reclaim will definitely help performance. I am sure once we get the thresholds correct, the performance will improve further.

```

>
> --- linux-2.6.23-rc8-mm2-cgkswapd/mm/memcontrol.c.BACKUP 2007-10-01
17:19:57.000000000 +0900
> +++ linux-2.6.23-rc8-mm2-cgkswapd/mm/memcontrol.c 2007-10-22 13:46:01.000000000 +0900
> @@ -27,6 +27,7 @@
> #include <linux/rcupdate.h>
> #include <linux/swap.h>
> #include <linux/spinlock.h>
> +#include <linux/workqueue.h>
> #include <linux/fs.h>
>
> #include <asm/uaccess.h>
> @@ -63,6 +64,8 @@ struct mem_cgroup {
>  */
>  spinlock_t lru_lock;
>  unsigned long control_type; /* control RSS or RSS+Pagecache */
> +
> + struct work_struct reclaim_work;
> };
>
> /*
> @@ -95,6 +98,9 @@ enum {
>
> static struct mem_cgroup init_mem_cgroup;
>
> +static DEFINE_MUTEX(mem_cgroup_workqueue_init_lock);
> +static struct workqueue_struct *mem_cgroup_workqueue;
> +
> static inline
> struct mem_cgroup *mem_cgroup_from_cont(struct cgroup *cont)
> {
> @@ -250,6 +256,69 @@ unsigned long mem_cgroup_isolate_pages(u
>  return nr_taken;
> }
>
>
> +static int

```

```

> +mem_cgroup_need_reclaim(struct mem_cgroup *mem)
> +{
> + struct res_counter * cnt = &mem->res;
> + int doreclaim;
> + unsigned long flags;
> +
> + /* XXX should be in res_counter */
> + /* XXX should not hardcode a watermark */
> + spin_lock_irqsave(&cnt->lock, flags);
> + doreclaim = cnt->usage > cnt->limit / 4 * 3;
> + spin_unlock_irqrestore(&cnt->lock, flags);
> +
> + return doreclaim;
> +}
> +
> +static void
> +mem_cgroup_schedule_reclaim_if_necessary(struct mem_cgroup *mem)
> +{
> +
> + if (mem_cgroup_workqueue == NULL) {
> + BUG_ON(mem->css.cgroup->parent != NULL);
> + return;
> + }
> +
> + if (work_pending(&mem->reclaim_work))
> + return;
> +
> + if (!mem_cgroup_need_reclaim(mem))
> + return;
> +
> + css_get(&mem->css); /* XXX need some thoughts wrt cgroup removal. */
> + /*
> + * XXX workqueue is not an ideal mechanism for our purpose.
> + * revisit later.
> + */
> + if (!queue_work(mem_cgroup_workqueue, &mem->reclaim_work))
> + css_put(&mem->css);
> +}
> +
> +static void
> +mem_cgroup_reclaim(struct work_struct *work)
> +{
> + struct mem_cgroup * const mem =
> + container_of(work, struct mem_cgroup, reclaim_work);
> + int batch_count = 128; /* XXX arbitrary */
> +
> + for (; batch_count > 0; batch_count--) {
> + if (!mem_cgroup_need_reclaim(mem))

```

```

> + break;
> + /*
> + * XXX try_to_free_foo is not a correct mechanism to
> + * use here. eg. ALLOCSTALL counter
> + * revisit later.
> + */
> + if (!try_to_free_mem_cgroup_pages(mem, GFP_KERNEL))
> + break;
> + }
> + if (batch_count == 0)
> + mem_cgroup_schedule_reclaim_if_necessary(mem);
> + css_put(&mem->css);
> +}
> +
> + /*
> + * Charge the memory controller for page usage.
> + * Return
> @@ -311,6 +380,9 @@ int mem_cgroup_charge(struct page *page,
> + */
> + while (res_counter_charge(&mem->res, PAGE_SIZE)) {
> + bool is_atomic = gfp_mask & GFP_ATOMIC;
> +
> + mem_cgroup_schedule_reclaim_if_necessary(mem);
> +
> + /*
> + * We cannot reclaim under GFP_ATOMIC, fail the charge
> + */
> @@ -551,8 +623,18 @@ mem_cgroup_create(struct cgroup_subsys *
> + if (unlikely((cont->parent) == NULL)) {
> + mem = &init_mem_cgroup;
> + init_mm.mem_cgroup = mem;
> - } else
> + } else {
> + /* XXX too late for the top-level cgroup */
> + if (mem_cgroup_workqueue == NULL) {
> + mutex_lock(&mem_cgroup_workqueue_init_lock);
> + if (mem_cgroup_workqueue == NULL) {
> + mem_cgroup_workqueue =
> + create_workqueue("mem_cgroup");
> + }
> + mutex_unlock(&mem_cgroup_workqueue_init_lock);
> + }
> + mem = kzalloc(sizeof(struct mem_cgroup), GFP_KERNEL);
> + }
> +
> + if (mem == NULL)
> + return NULL;
> @@ -562,6 +644,7 @@ mem_cgroup_create(struct cgroup_subsys *

```

```

> INIT_LIST_HEAD(&mem->inactive_list);
> spin_lock_init(&mem->lru_lock);
> mem->control_type = MEM_CGROUP_TYPE_ALL;
> + INIT_WORK(&mem->reclaim_work, mem_cgroup_reclaim);
> return &mem->css;
> }
>

```

Can we potentially avoid direct reclaim and sleep if the charge is exceeded and let the background reclaim do the job and wake us up?

--Vaidy

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

Subject: Re: [RFC] [PATCH] memory controller background reclamation
Posted by [yamamoto](#) on Mon, 22 Oct 2007 23:44:30 GMT
[View Forum Message](#) <> [Reply to Message](#)

hi,

```

> > @@ -250,6 +256,69 @@ unsigned long mem_cgroup_isolate_pages(u
> > return nr_taken;
> > }
> >
> > +static int
> > +mem_cgroup_need_reclaim(struct mem_cgroup *mem)
> > +{
> > + struct res_counter * const cnt = &mem->res;
> > + int doreclaim;
> > + unsigned long flags;
> > +
> > + /* XXX should be in res_counter */
> > + /* XXX should not hardcode a watermark */
> >
> > We could add the following API to resource counters
> >
> > res_counter_set_low_watermark
> > res_counter_set_high_watermark
> > res_counter_below_low_watermark
> > res_counter_above_high_watermark
> >
> > and add
> >

```

> low_watermark
> high_watermark
>
> members to the resource group. We could push out data
> upto the low watermark from the cgroup.

it sounds fine to me.

```
> > +static void
> > +mem_cgroup_reclaim(struct work_struct *work)
> > +{
> > + struct mem_cgroup * const mem =
> > +     container_of(work, struct mem_cgroup, reclaim_work);
> > + int batch_count = 128; /* XXX arbitrary */
> > +
> > + for (; batch_count > 0; batch_count--) {
> > +     if (!mem_cgroup_need_reclaim(mem))
> > +         break;
> > + /*
> > +  * XXX try_to_free_foo is not a correct mechanism to
> > +  * use here. eg. ALLOCSTALL counter
> > +  * revisit later.
> > +  */
> > + if (!try_to_free_mem_cgroup_pages(mem, GFP_KERNEL))
>
> We could make try_to_free_mem_cgroup_pages, batch aware and pass that
> in scan_control.
```

in the comment above, i meant that it might be better to introduce something like balance_pgdat rather than using try_to_free_mem_cgroup_pages. with the current design of cgroup lru lists, probably it doesn't matter much except statistics, tho.

YAMAMOTO Takashi

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

Subject: Re: [RFC] [PATCH] memory controller background reclamation
Posted by [yamamoto](#) on Mon, 22 Oct 2007 23:45:23 GMT
[View Forum Message](#) <> [Reply to Message](#)

hi,

> Can we potentially avoid direct reclaim and sleep if the charge is exceeded
> and let the background reclaim do the job and wake us up?

yes, but i left it for now because direct reclaim has its own benefits.

YAMAMOTO Takashi

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

Subject: Re: [RFC] [PATCH] memory controller background reclamation
Posted by [yamamoto](#) on Thu, 15 Nov 2007 06:16:02 GMT

[View Forum Message](#) <> [Reply to Message](#)

> > We could add the following API to resource counters
> >
> > res_counter_set_low_watermark
> > res_counter_set_high_watermark
> > res_counter_below_low_watermark
> > res_counter_above_high_watermark
> >
> > and add
> >
> > low_watermark
> > high_watermark
> >
> > members to the resource group. We could push out data
> > upto the low watermark from the cgroup.

i implemented something like that. (and rebased to 2.6.24-rc2-mm1.)

what's the best way to expose watermarks to userland is an open question.
i took the simplest way for now. do you have any suggestions?

YAMAMOTO Takashi

```
--- ./include/linux/res_counter.h.orig 2007-11-14 15:57:31.000000000 +0900
+++ ./include/linux/res_counter.h 2007-11-14 16:03:24.000000000 +0900
@@ -32,6 +32,13 @@ struct res_counter {
     * the number of unsuccessful attempts to consume the resource
     */
     unsigned long long failcnt;
+
+ /*
+  * watermarks
+  */
+ unsigned long long high_watermark;
```

```

+ unsigned long long low_watermark;
+
+ /*
+  * the lock to protect all of the above.
+  * the routines below consider this to be IRQ-safe
@@ -66,6 +73,8 @@ enum {
    RES_USAGE,
    RES_LIMIT,
    RES_FAILCNT,
+ RES_HIGH_WATERMARK,
+ RES_LOW_WATERMARK,
};

+ /*
@@ -124,4 +133,26 @@ static inline bool res_counter_check_and
    return ret;
}

+static inline bool res_counter_below_low_watermark(struct res_counter *cnt)
+{
+ bool ret;
+ unsigned long flags;
+
+ spin_lock_irqsave(&cnt->lock, flags);
+ ret = cnt->usage < cnt->low_watermark;
+ spin_unlock_irqrestore(&cnt->lock, flags);
+ return ret;
+}
+
+static inline bool res_counter_above_high_watermark(struct res_counter *cnt)
+{
+ bool ret;
+ unsigned long flags;
+
+ spin_lock_irqsave(&cnt->lock, flags);
+ ret = cnt->usage > cnt->high_watermark;
+ spin_unlock_irqrestore(&cnt->lock, flags);
+ return ret;
+}
+
+ #endif
--- ./kernel/res_counter.c.orig 2007-11-14 15:57:31.000000000 +0900
+++ ./kernel/res_counter.c 2007-11-14 16:03:24.000000000 +0900
@@ -17,6 +17,8 @@ void res_counter_init(struct res_counter
{
    spin_lock_init(&counter->lock);
    counter->limit = (unsigned long long)LLONG_MAX;
+ counter->high_watermark = (unsigned long long)LLONG_MAX;

```

```

+ counter->low_watermark = (unsigned long long)LLONG_MAX;
}

int res_counter_charge_locked(struct res_counter *counter, unsigned long val)
@@ -69,6 +71,10 @@ res_counter_member(struct res_counter *c
    return &counter->limit;
    case RES_FAILCNT:
        return &counter->failcnt;
+ case RES_HIGH_WATERMARK:
+ return &counter->high_watermark;
+ case RES_LOW_WATERMARK:
+ return &counter->low_watermark;
};

BUG();
@@ -99,6 +105,7 @@ ssize_t res_counter_write(struct res_cou
int ret;
char *buf, *end;
unsigned long long tmp, *val;
+ unsigned long flags;

    buf = kmalloc(nbytes + 1, GFP_KERNEL);
    ret = -ENOMEM;
@@ -122,9 +129,29 @@ ssize_t res_counter_write(struct res_cou
    goto out_free;
}

+ spin_lock_irqsave(&counter->lock, flags);
+ val = res_counter_member(counter, member);
+ /* ensure low_watermark <= high_watermark <= limit */
+ switch (member) {
+ case RES_LIMIT:
+ if (tmp < counter->high_watermark)
+ goto out_locked;
+ break;
+ case RES_HIGH_WATERMARK:
+ if (tmp > counter->limit || tmp < counter->low_watermark)
+ goto out_locked;
+ break;
+ case RES_LOW_WATERMARK:
+ if (tmp > counter->high_watermark)
+ goto out_locked;
+ break;
+ }
+ *val = tmp;
+ BUG_ON(counter->high_watermark > counter->limit);
+ BUG_ON(counter->low_watermark > counter->high_watermark);
    ret = nbytes;

```

```

+out_locked:
+ spin_unlock_irqrestore(&counter->lock, flags);
out_free:
    kfree(buf);
out:
--- ./mm/memcontrol.c.orig 2007-11-14 15:57:46.000000000 +0900
+++ ./mm/memcontrol.c 2007-11-14 16:03:53.000000000 +0900
@@ -28,6 +28,7 @@
#include <linux/rcupdate.h>
#include <linux/swap.h>
#include <linux/spinlock.h>
+#include <linux/workqueue.h>
#include <linux/fs.h>
#include <linux/seq_file.h>

@@ -110,6 +111,10 @@ struct mem_cgroup {
    * statistics.
    */
    struct mem_cgroup_stat stat;
+ /*
+  * background reclamation.
+  */
+ struct work_struct reclaim_work;
};

/*
@@ -168,6 +173,9 @@ static void mem_cgroup_charge_statistics

static struct mem_cgroup init_mem_cgroup;

+static DEFINE_MUTEX(mem_cgroup_workqueue_init_lock);
+static struct workqueue_struct *mem_cgroup_workqueue;
+
static inline
struct mem_cgroup *mem_cgroup_from_cont(struct cgroup *cont)
{
@@ -375,6 +383,50 @@ unsigned long mem_cgroup_isolate_pages(u
    return nr_taken;
}

+static void
+mem_cgroup_schedule_reclaim(struct mem_cgroup *mem)
+{
+
+ if (mem_cgroup_workqueue == NULL) {
+     BUG_ON(mem->css.cgroup->parent != NULL);
+     return;
+ }

```

```

+
+ if (work_pending(&mem->reclaim_work))
+ return;
+
+ css_get(&mem->css); /* XXX need some thoughts wrt cgroup removal. */
+ /*
+  * XXX workqueue is not an ideal mechanism for our purpose.
+  * revisit later.
+  */
+ if (!queue_work(mem_cgroup_workqueue, &mem->reclaim_work))
+ css_put(&mem->css);
+}
+
+static void
+mem_cgroup_reclaim(struct work_struct *work)
+{
+ struct mem_cgroup * const mem =
+ container_of(work, struct mem_cgroup, reclaim_work);
+ int batch_count = 128; /* XXX arbitrary */
+
+ for (; batch_count > 0; batch_count--) {
+ if (res_counter_below_low_watermark(&mem->res))
+ break;
+ /*
+  * XXX try_to_free_foo is not a correct mechanism to
+  * use here. eg. ALLOCSTALL counter
+  * revisit later.
+  */
+ if (!try_to_free_mem_cgroup_pages(mem, GFP_KERNEL))
+ break;
+ }
+ if (batch_count == 0)
+ mem_cgroup_schedule_reclaim(mem);
+ css_put(&mem->css);
+}
+
+/*
+ * Charge the memory controller for page usage.
+ * Return
+@@ -439,11 +491,18 @@ retry:
+ rcu_read_unlock();
+
+/*
+ * schedule background reclaim if we are above the high watermark.
+ */
+ if (res_counter_above_high_watermark(&mem->res))
+ mem_cgroup_schedule_reclaim(mem);
+

```

```

+ /*
+  * If we created the page_cgroup, we should free it on exceeding
+  * the cgroup limit.
+  */
while (res_counter_charge(&mem->res, PAGE_SIZE)) {
    bool is_atomic = gfp_mask & GFP_ATOMIC;
+
+    /*
+     * We cannot reclaim under GFP_ATOMIC, fail the charge
+     */
@@ -861,6 +920,18 @@ static struct cftype mem_cgroup_files[]
    .read = mem_cgroup_read,
    },
    {
+ .name = "high_watermark_in_bytes",
+ .private = RES_HIGH_WATERMARK,
+ .write = mem_cgroup_write,
+ .read = mem_cgroup_read,
+ },
+ {
+ .name = "low_watermark_in_bytes",
+ .private = RES_LOW_WATERMARK,
+ .write = mem_cgroup_write,
+ .read = mem_cgroup_read,
+ },
+ {
    .name = "control_type",
    .write = mem_control_type_write,
    .read = mem_control_type_read,
@@ -886,8 +957,18 @@ mem_cgroup_create(struct cgroup_subsys *
    if (unlikely((cont->parent) == NULL)) {
        mem = &init_mem_cgroup;
        init_mm.mem_cgroup = mem;
- } else
+ } else {
+ /* XXX too late for the top-level cgroup */
+ if (mem_cgroup_workqueue == NULL) {
+     mutex_lock(&mem_cgroup_workqueue_init_lock);
+     if (mem_cgroup_workqueue == NULL) {
+         mem_cgroup_workqueue =
+             create_workqueue("mem_cgroup");
+     }
+     mutex_unlock(&mem_cgroup_workqueue_init_lock);
+ }
    mem = kzalloc(sizeof(struct mem_cgroup), GFP_KERNEL);
+ }

    if (mem == NULL)

```

```
    return NULL;
@@ -897,6 +978,7 @@ mem_cgroup_create(struct cgroup_subsys *
    INIT_LIST_HEAD(&mem->inactive_list);
    spin_lock_init(&mem->lru_lock);
    mem->control_type = MEM_CGROUP_TYPE_ALL;
+ INIT_WORK(&mem->reclaim_work, mem_cgroup_reclaim);
    return &mem->css;
}
```

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

Subject: Re: [RFC] [PATCH] memory controller background reclamation
Posted by [yamamoto](#) on Thu, 22 Nov 2007 08:57:33 GMT
[View Forum Message](#) <> [Reply to Message](#)

```
> > > We could add the following API to resource counters
> > >
> > > res_counter_set_low_watermark
> > > res_counter_set_high_watermark
> > > res_counter_below_low_watermark
> > > res_counter_above_high_watermark
> > >
> > > and add
> > >
> > > low_watermark
> > > high_watermark
> > >
> > > members to the resource group. We could push out data
> > > upto the low watermark from the cgroup.
>
> i implemented something like that. (and rebased to 2.6.24-rc2-mm1.)
>
> what's the best way to expose watermarks to userland is an open question.
> i took the simplest way for now. do you have any suggestions?
>
> YAMAMOTO Takashi
```

here's a new version.

changes from the previous:

- rebase to linux-2.6.24-rc2-mm1 + kamezawa patchset.
- create workqueue when setting high watermark, rather than cgroup creation which is earlier on boot.

YAMAMOTO Takashi

Signed-off-by: YAMAMOTO Takashi <yamamoto@valinux.co.jp>

```
---
--- linux-2.6.24-rc2-mm1-kame-pd/include/linux/res_counter.h.BACKUP 2007-11-14
16:05:48.000000000 +0900
+++ linux-2.6.24-rc2-mm1-kame-pd/include/linux/res_counter.h 2007-11-22 15:14:32.000000000
+0900
@@ -32,6 +32,13 @@ struct res_counter {
    * the number of unsuccessful attempts to consume the resource
    */
    unsigned long long failcnt;
+
+ /*
+  * watermarks
+  */
+ unsigned long long high_watermark;
+ unsigned long long low_watermark;
+
+ /*
+  * the lock to protect all of the above.
+  * the routines below consider this to be IRQ-safe
@@ -66,6 +73,8 @@ enum {
    RES_USAGE,
    RES_LIMIT,
    RES_FAILCNT,
+ RES_HIGH_WATERMARK,
+ RES_LOW_WATERMARK,
};

/*
@@ -124,4 +133,26 @@ static inline bool res_counter_check_and
    return ret;
}

+static inline bool res_counter_below_low_watermark(struct res_counter *cnt)
+{
+ bool ret;
+ unsigned long flags;
+
+ spin_lock_irqsave(&cnt->lock, flags);
+ ret = cnt->usage < cnt->low_watermark;
+ spin_unlock_irqrestore(&cnt->lock, flags);
+ return ret;
+}
+
```

```

+static inline bool res_counter_above_high_watermark(struct res_counter *cnt)
+{
+ bool ret;
+ unsigned long flags;
+
+ spin_lock_irqsave(&cnt->lock, flags);
+ ret = cnt->usage > cnt->high_watermark;
+ spin_unlock_irqrestore(&cnt->lock, flags);
+ return ret;
+}
+
#endif
--- linux-2.6.24-rc2-mm1-kame-pd/kernel/res_counter.c.BACKUP 2007-11-14
16:05:52.000000000 +0900
+++ linux-2.6.24-rc2-mm1-kame-pd/kernel/res_counter.c 2007-11-22 15:14:32.000000000 +0900
@@ -17,6 +17,8 @@ void res_counter_init(struct res_counter
{
    spin_lock_init(&counter->lock);
    counter->limit = (unsigned long long)LLONG_MAX;
+ counter->high_watermark = (unsigned long long)LLONG_MAX;
+ counter->low_watermark = (unsigned long long)LLONG_MAX;
}

int res_counter_charge_locked(struct res_counter *counter, unsigned long val)
@@ -69,6 +71,10 @@ res_counter_member(struct res_counter *c
    return &counter->limit;
    case RES_FAILCNT:
        return &counter->failcnt;
+ case RES_HIGH_WATERMARK:
+ return &counter->high_watermark;
+ case RES_LOW_WATERMARK:
+ return &counter->low_watermark;
};

BUG();
@@ -99,6 +105,7 @@ ssize_t res_counter_write(struct res_cou
int ret;
char *buf, *end;
unsigned long long tmp, *val;
+ unsigned long flags;

    buf = kmalloc(nbytes + 1, GFP_KERNEL);
    ret = -ENOMEM;
@@ -122,9 +129,29 @@ ssize_t res_counter_write(struct res_cou
    goto out_free;
}

+ spin_lock_irqsave(&counter->lock, flags);

```

```

    val = res_counter_member(counter, member);
+ /* ensure low_watermark <= high_watermark <= limit */
+ switch (member) {
+ case RES_LIMIT:
+ if (tmp < counter->high_watermark)
+ goto out_locked;
+ break;
+ case RES_HIGH_WATERMARK:
+ if (tmp > counter->limit || tmp < counter->low_watermark)
+ goto out_locked;
+ break;
+ case RES_LOW_WATERMARK:
+ if (tmp > counter->high_watermark)
+ goto out_locked;
+ break;
+ }
    *val = tmp;
+ BUG_ON(counter->high_watermark > counter->limit);
+ BUG_ON(counter->low_watermark > counter->high_watermark);
    ret = nbytes;
+out_locked:
+ spin_unlock_irqrestore(&counter->lock, flags);
    out_free:
    kfree(buf);
    out:
--- linux-2.6.24-rc2-mm1-kame-pd/mm/memcontrol.c.BACKUP 2007-11-20 13:11:09.000000000
+0900
+++ linux-2.6.24-rc2-mm1-kame-pd/mm/memcontrol.c 2007-11-22 16:29:26.000000000 +0900
@@ -28,6 +28,7 @@
#include <linux/rcupdate.h>
#include <linux/swap.h>
#include <linux/spinlock.h>
+#include <linux/workqueue.h>
#include <linux/fs.h>
#include <linux/seq_file.h>

@@ -138,6 +139,10 @@ struct mem_cgroup {
    * statistics.
    */
    struct mem_cgroup_stat stat;
+ /*
+  * background reclamation.
+  */
+ struct work_struct reclaim_work;
};

/*
@@ -240,6 +245,21 @@ static unsigned long mem_cgroup_get_all_

```

```

static struct mem_cgroup init_mem_cgroup;

+static DEFINE_MUTEX(mem_cgroup_workqueue_init_lock);
+static struct workqueue_struct *mem_cgroup_workqueue;
+
+static void mem_cgroup_create_workqueue(void)
+{
+
+ if (mem_cgroup_workqueue != NULL)
+ return;
+
+ mutex_lock(&mem_cgroup_workqueue_init_lock);
+ if (mem_cgroup_workqueue == NULL)
+ mem_cgroup_workqueue = create_workqueue("mem_cgroup");
+ mutex_unlock(&mem_cgroup_workqueue_init_lock);
+}
+
+static inline
+struct mem_cgroup *mem_cgroup_from_cont(struct cgroup *cont)
+{
@@ -566,6 +586,50 @@ unsigned long mem_cgroup_isolate_pages(u
return nr_taken;
}

+static void
+mem_cgroup_schedule_reclaim(struct mem_cgroup *mem)
+{
+
+ if (mem_cgroup_workqueue == NULL) {
+ BUG_ON(mem->css.cgroup->parent != NULL);
+ return;
+ }
+
+ if (work_pending(&mem->reclaim_work))
+ return;
+
+ css_get(&mem->css); /* XXX need some thoughts wrt cgroup removal. */
+ /*
+ * XXX workqueue is not an ideal mechanism for our purpose.
+ * revisit later.
+ */
+ if (!queue_work(mem_cgroup_workqueue, &mem->reclaim_work))
+ css_put(&mem->css);
+}
+
+static void
+mem_cgroup_reclaim(struct work_struct *work)

```

```

+{
+ struct mem_cgroup * const mem =
+   container_of(work, struct mem_cgroup, reclaim_work);
+ int batch_count = 128; /* XXX arbitrary */
+
+ for (; batch_count > 0; batch_count--) {
+   if (res_counter_below_low_watermark(&mem->res))
+     break;
+   /*
+    * XXX try_to_free_foo is not a correct mechanism to
+    * use here. eg. ALLOCSTALL counter
+    * revisit later.
+    */
+   if (!try_to_free_mem_cgroup_pages(mem, GFP_KERNEL))
+     break;
+ }
+ if (batch_count == 0)
+   mem_cgroup_schedule_reclaim(mem);
+ css_put(&mem->css);
+}
+
+/*
+ * Charge the memory controller for page usage.
+ * Return
@@ -631,6 +695,12 @@ retry:
   rcu_read_unlock();

+/*
+ * schedule background reclaim if we are above the high watermark.
+ */
+ if (res_counter_above_high_watermark(&mem->res))
+   mem_cgroup_schedule_reclaim(mem);
+
+/*
+ * If we created the page_cgroup, we should free it on exceeding
+ * the cgroup limit.
+ */
@@ -939,9 +1009,16 @@ static ssize_t mem_cgroup_write(struct c
   struct file *file, const char __user *userbuf,
   size_t nbytes, loff_t *ppos)
+ {
+   - return res_counter_write(&mem_cgroup_from_cont(cont)->res,
+   + ssize_t ret;
+
+   + ret = res_counter_write(&mem_cgroup_from_cont(cont)->res,
+     cft->private, userbuf, nbytes, ppos,
+     mem_cgroup_write_strategy);
+
+

```

```

+ if (ret >= 0 && cft->private == RES_HIGH_WATERMARK)
+ mem_cgroup_create_workqueue();
+
+ return ret;
+ }

static ssize_t mem_control_type_write(struct cgroup *cont,
@@ -1097,6 +1174,18 @@ static struct cftype mem_cgroup_files[]
    .read = mem_cgroup_read,
    },
    {
+ .name = "high_watermark_in_bytes",
+ .private = RES_HIGH_WATERMARK,
+ .write = mem_cgroup_write,
+ .read = mem_cgroup_read,
+ },
+ {
+ .name = "low_watermark_in_bytes",
+ .private = RES_LOW_WATERMARK,
+ .write = mem_cgroup_write,
+ .read = mem_cgroup_read,
+ },
+ {
    .name = "control_type",
    .write = mem_control_type_write,
    .read = mem_control_type_read,
@@ -1161,6 +1250,8 @@ mem_cgroup_create(struct cgroup_subsys *
    if (alloc_mem_cgroup_per_zone_info(mem, node))
        goto free_out;

+ INIT_WORK(&mem->reclaim_work, mem_cgroup_reclaim);
+
    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: Re: [RFC] [PATCH] memory controller background reclamation
Posted by [KAMEZAWA Hiroyuki](#) on Thu, 22 Nov 2007 13:33:13 GMT
[View Forum Message](#) <> [Reply to Message](#)

>here's a new version.

>

Thank you. I'll review and try to merge yours in the next week,

Regards,
-Kame

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

Subject: Re: [RFC] [PATCH] memory controller background reclamation
Posted by [Balbir Singh](#) on Fri, 23 Nov 2007 03:46:12 GMT
[View Forum Message](#) <> [Reply to Message](#)

YAMAMOTO Takashi wrote:
> changes from the previous:
> - rebase to linux-2.6.24-rc2-mm1 + kamezawa patchset.
> - create workqueue when setting high watermark, rather than cgroup creation
> which is earlier on boot.
>
> YAMAMOTO Takashi
>
>
> Signed-off-by: YAMAMOTO Takashi <yamamoto@valinux.co.jp>
> ---
>
> --- linux-2.6.24-rc2-mm1-kame-pd/include/linux/res_counter.h.BACKUP 2007-11-14
16:05:48.000000000 +0900
> +++ linux-2.6.24-rc2-mm1-kame-pd/include/linux/res_counter.h 2007-11-22
15:14:32.000000000 +0900
> @@ -32,6 +32,13 @@ struct res_counter {
> * the number of unsuccessful attempts to consume the resource
> */
> unsigned long long failcnt;
> +
> + /*
> + * watermarks
> + */
> + unsigned long long high_watermark;
> + unsigned long long low_watermark;
> +
> + /*
> * the lock to protect all of the above.
> * the routines below consider this to be IRQ-safe
> @@ -66,6 +73,8 @@ enum {
> RES_USAGE,
> RES_LIMIT,
> RES_FAILCNT,

```

> + RES_HIGH_WATERMARK,
> + RES_LOW_WATERMARK,
> };
>
> /*
> @@ -124,4 +133,26 @@ static inline bool res_counter_check_und
> return ret;
> }
>
> +static inline bool res_counter_below_low_watermark(struct res_counter *cnt)
> +{
> + bool ret;
> + unsigned long flags;
> +
> + spin_lock_irqsave(&cnt->lock, flags);
> + ret = cnt->usage < cnt->low_watermark;
> + spin_unlock_irqrestore(&cnt->lock, flags);
> + return ret;
> +}
> +
> +static inline bool res_counter_above_high_watermark(struct res_counter *cnt)
> +{
> + bool ret;
> + unsigned long flags;
> +
> + spin_lock_irqsave(&cnt->lock, flags);
> + ret = cnt->usage > cnt->high_watermark;
> + spin_unlock_irqrestore(&cnt->lock, flags);
> + return ret;
> +}
> +
> #endif
> --- linux-2.6.24-rc2-mm1-kame-pd/kernel/res_counter.c.BACKUP 2007-11-14
16:05:52.000000000 +0900
> +++ linux-2.6.24-rc2-mm1-kame-pd/kernel/res_counter.c 2007-11-22 15:14:32.000000000
+0900
> @@ -17,6 +17,8 @@ void res_counter_init(struct res_counter
> {
> spin_lock_init(&counter->lock);
> counter->limit = (unsigned long long)LLONG_MAX;
> + counter->high_watermark = (unsigned long long)LLONG_MAX;
> + counter->low_watermark = (unsigned long long)LLONG_MAX;

```

Should low watermark also be LLONG_MAX?

```

> }
>
> int res_counter_charge_locked(struct res_counter *counter, unsigned long val)

```

```

> @@ -69,6 +71,10 @@ res_counter_member(struct res_counter *c
>     return &counter->limit;
>     case RES_FAILCNT:
>         return &counter->failcnt;
> + case RES_HIGH_WATERMARK:
> +     return &counter->high_watermark;
> + case RES_LOW_WATERMARK:
> +     return &counter->low_watermark;
>     };
>
>     BUG();
> @@ -99,6 +105,7 @@ ssize_t res_counter_write(struct res_cou
>     int ret;
>     char *buf, *end;
>     unsigned long long tmp, *val;
> + unsigned long flags;
>
>     buf = kmalloc(nbytes + 1, GFP_KERNEL);
>     ret = -ENOMEM;
> @@ -122,9 +129,29 @@ ssize_t res_counter_write(struct res_cou
>     goto out_free;
> }
>
> + spin_lock_irqsave(&counter->lock, flags);
>     val = res_counter_member(counter, member);
> + /* ensure low_watermark <= high_watermark <= limit */
> + switch (member) {
> + case RES_LIMIT:
> +     if (tmp < counter->high_watermark)
> +         goto out_locked;
> +     break;
> + case RES_HIGH_WATERMARK:
> +     if (tmp > counter->limit || tmp < counter->low_watermark)
> +         goto out_locked;
> +     break;
> + case RES_LOW_WATERMARK:
> +     if (tmp > counter->high_watermark)
> +         goto out_locked;
> +     break;
> + }
>     *val = tmp;
> + BUG_ON(counter->high_watermark > counter->limit);
> + BUG_ON(counter->low_watermark > counter->high_watermark);
>     ret = nbytes;
> +out_locked:
> + spin_unlock_irqrestore(&counter->lock, flags);
>     out_free:
>     kfree(buf);

```

```

> out:
> --- linux-2.6.24-rc2-mm1-kame-pd/mm/memcontrol.c.BACKUP 2007-11-20 13:11:09.000000000
+0900
> +++ linux-2.6.24-rc2-mm1-kame-pd/mm/memcontrol.c 2007-11-22 16:29:26.000000000 +0900
> @@ -28,6 +28,7 @@
> #include <linux/rcupdate.h>
> #include <linux/swap.h>
> #include <linux/spinlock.h>
> +#include <linux/workqueue.h>
> #include <linux/fs.h>
> #include <linux/seq_file.h>
>
> @@ -138,6 +139,10 @@ struct mem_cgroup {
> * statistics.
> */
> struct mem_cgroup_stat stat;
> + /*
> + * background reclamation.
> + */
> + struct work_struct reclaim_work;
> };
>
> /*
> @@ -240,6 +245,21 @@ static unsigned long mem_cgroup_get_all_
>
> static struct mem_cgroup init_mem_cgroup;
>
> +static DEFINE_MUTEX(mem_cgroup_workqueue_init_lock);
> +static struct workqueue_struct *mem_cgroup_workqueue;
> +
> +static void mem_cgroup_create_workqueue(void)
> +{
> +
> + if (mem_cgroup_workqueue != NULL)
> + return;
> +
> + mutex_lock(&mem_cgroup_workqueue_init_lock);
> + if (mem_cgroup_workqueue == NULL)
> + mem_cgroup_workqueue = create_workqueue("mem_cgroup");
> + mutex_unlock(&mem_cgroup_workqueue_init_lock);
> +}
> +
> static inline
> struct mem_cgroup *mem_cgroup_from_cont(struct cgroup *cont)
> {
> @@ -566,6 +586,50 @@ unsigned long mem_cgroup_isolate_pages(u
> return nr_taken;
> }

```

```

>
> +static void
> +mem_cgroup_schedule_reclaim(struct mem_cgroup *mem)
> +{
> +
> + if (mem_cgroup_workqueue == NULL) {
> + BUG_ON(mem->css.cgroup->parent != NULL);
> + return;
> + }
> +
> + if (work_pending(&mem->reclaim_work))
> + return;
> +
> + css_get(&mem->css); /* XXX need some thoughts wrt cgroup removal. */
> + /*
> + * XXX workqueue is not an ideal mechanism for our purpose.
> + * revisit later.
> + */
> + if (!queue_work(mem_cgroup_workqueue, &mem->reclaim_work))
> + css_put(&mem->css);
> +}
> +
> +static void
> +mem_cgroup_reclaim(struct work_struct *work)
> +{
> + struct mem_cgroup * const mem =
> + container_of(work, struct mem_cgroup, reclaim_work);
> + int batch_count = 128; /* XXX arbitrary */

```

Could we define and use something like MEM_CGROUP_BATCH_COUNT for now?
 Later we could consider and see if it needs to be tunable. numbers are
 hard to read in code.

```

> +
> + for (; batch_count > 0; batch_count--) {
> + if (res_counter_below_low_watermark(&mem->res))
> + break;

```

Shouldn't we also check to see that we start reclaim in background only
 when we are above the high watermark?

```

> + /*
> + * XXX try_to_free_foo is not a correct mechanism to
> + * use here. eg. ALLOCSTALL counter
> + * revisit later.
> + */
> + if (!try_to_free_mem_cgroup_pages(mem, GFP_KERNEL))
> + break;

```

```

> + }
> + if (batch_count == 0)
> + mem_cgroup_schedule_reclaim(mem);
> + css_put(&mem->css);
> +}
> +
> /*
>  * Charge the memory controller for page usage.
>  * Return
> @@ -631,6 +695,12 @@ retry:
>  rcu_read_unlock();
>
> /*
> + * schedule background reclaim if we are above the high watermark.
> + */
> + if (res_counter_above_high_watermark(&mem->res))
> + mem_cgroup_schedule_reclaim(mem);
> +
> + /*
>  * If we created the page_cgroup, we should free it on exceeding
>  * the cgroup limit.
>  */
> @@ -939,9 +1009,16 @@ static ssize_t mem_cgroup_write(struct c
>  struct file *file, const char __user *userbuf,
>  size_t nbytes, loff_t *ppos)
> {
> - return res_counter_write(&mem_cgroup_from_cont(cont)->res,
> + ssize_t ret;
> +
> + ret = res_counter_write(&mem_cgroup_from_cont(cont)->res,
>  cft->private, userbuf, nbytes, ppos,
>  mem_cgroup_write_strategy);
> +
> + if (ret >= 0 && cft->private == RES_HIGH_WATERMARK)
> + mem_cgroup_create_workqueue();
> +
> + return ret;
> }
>
> static ssize_t mem_control_type_write(struct cgroup *cont,
> @@ -1097,6 +1174,18 @@ static struct cftype mem_cgroup_files[]
>  .read = mem_cgroup_read,
>  },
>  {
> + .name = "high_watermark_in_bytes",
> + .private = RES_HIGH_WATERMARK,
> + .write = mem_cgroup_write,
> + .read = mem_cgroup_read,

```

```

> + },
> + {
> + .name = "low_watermark_in_bytes",
> + .private = RES_LOW_WATERMARK,
> + .write = mem_cgroup_write,
> + .read = mem_cgroup_read,
> + },
> + {
> .name = "control_type",
> .write = mem_control_type_write,
> .read = mem_control_type_read,
> @@ -1161,6 +1250,8 @@ mem_cgroup_create(struct cgroup_subsys *
> if (alloc_mem_cgroup_per_zone_info(mem, node))
> goto free_out;
>
> + INIT_WORK(&mem->reclaim_work, mem_cgroup_reclaim);
> +
> return &mem->css;
> free_out:
> for_each_node_state(node, N_POSSIBLE)

```

I'll start some tests on these patches.

--

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] [PATCH] memory controller background reclamation
Posted by [yamamoto](#) on Mon, 26 Nov 2007 02:47:10 GMT
[View Forum Message](#) <> [Reply to Message](#)

hi,

```

> > --- linux-2.6.24-rc2-mm1-kame-pd/kernel/res_counter.c.BACKUP 2007-11-14
16:05:52.000000000 +0900
> > +++ linux-2.6.24-rc2-mm1-kame-pd/kernel/res_counter.c 2007-11-22 15:14:32.000000000
+0900
> > @@ -17,6 +17,8 @@ void res_counter_init(struct res_counter
> > {

```

```
> > spin_lock_init(&counter->lock);
> > counter->limit = (unsigned long long)LLONG_MAX;
> > + counter->high_watermark = (unsigned long long)LLONG_MAX;
> > + counter->low_watermark = (unsigned long long)LLONG_MAX;
>
> Should low watermark also be LLONG_MAX?
```

what else do you suggest? 0?

currently it doesn't matter much because low_watermark is not used at all as far as high_watermark is LLONG_MAX.

```
> > +static void
> > +mem_cgroup_reclaim(struct work_struct *work)
> > +{
> > + struct mem_cgroup * const mem =
> > +     container_of(work, struct mem_cgroup, reclaim_work);
> > + int batch_count = 128; /* XXX arbitrary */
>
> Could we define and use something like MEM_CGROUP_BATCH_COUNT for now?
> Later we could consider and see if it needs to be tunable. numbers are
> hard to read in code.
```

although i don't think it makes sense, i can do so if you prefer.

```
> > +
> > + for (; batch_count > 0; batch_count--) {
> > + if (res_counter_below_low_watermark(&mem->res))
> > +     break;
>
> Shouldn't we also check to see that we start reclaim in background only
> when we are above the high watermark?
```

i don't understand what you mean. can you explain?

highwatermark is checked by mem_cgroup_charge_common before waking these threads.

> I'll start some tests on these patches.

thanks.

YAMAMOTO Takashi

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

Subject: Re: [RFC] [PATCH] memory controller background reclamation

Posted by [Balbir Singh](#) on Mon, 26 Nov 2007 03:00:19 GMT

[View Forum Message](#) <> [Reply to Message](#)

YAMAMOTO Takashi wrote:

> hi,

>

>>> --- linux-2.6.24-rc2-mm1-kame-pd/kernel/res_counter.c.BACKUP 2007-11-14

16:05:52.000000000 +0900

>>> +++ linux-2.6.24-rc2-mm1-kame-pd/kernel/res_counter.c 2007-11-22 15:14:32.000000000

+0900

>>> @@ -17,6 +17,8 @@ void res_counter_init(struct res_counter

>>> {

>>> spin_lock_init(&counter->lock);

>>> counter->limit = (unsigned long long)LLONG_MAX;

>>> + counter->high_watermark = (unsigned long long)LLONG_MAX;

>>> + counter->low_watermark = (unsigned long long)LLONG_MAX;

>> Should low watermark also be LLONG_MAX?

>

> what else do you suggest? 0?

Something invalid or a good default value. I think LLONG_MAX

is good for now, that ensures that no background reclaim

happens till the administrator sets it up.

> currently it doesn't matter much because low_watermark is not used at all

> as far as high_watermark is LLONG_MAX.

>

Don't we use by checking res_counter_below_low_watermark()?

>>> +static void

>>> +mem_cgroup_reclaim(struct work_struct *work)

>>> +{

>>> + struct mem_cgroup * const mem =

>>> + container_of(work, struct mem_cgroup, reclaim_work);

>>> + int batch_count = 128; /* XXX arbitrary */

>> Could we define and use something like MEM_CGROUP_BATCH_COUNT for now?

>> Later we could consider and see if it needs to be tunable. numbers are

>> hard to read in code.

>

> although i don't think it makes sense, i can do so if you prefer.

>

Using numbers like 128 make the code unreadable. I prefer something like MEM_CGROUP_BATCH_COUNT since its more readable than 128. If we ever propagate batch_count to other dependent functions, I'd much rather do it with a well defined name.

```
>>> +
>>> + for (; batch_count > 0; batch_count--) {
>>> + if (res_counter_below_low_watermark(&mem->res))
>>> + break;
>> Shouldn't we also check to see that we start reclaim in background only
>> when we are above the high watermark?
>
> i don't understand what you mean. can you explain?
> highwatermark is checked by mem_cgroup_charge_common before waking
> these threads.
>
```

OK, that clarifies

```
>> I'll start some tests on these patches.
>
> thanks.
>
> YAMAMOTO Takashi
```

--

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] [PATCH] memory controller background reclamation
Posted by [yamamoto](#) on Mon, 26 Nov 2007 03:12:53 GMT
[View Forum Message](#) <> [Reply to Message](#)

```
> > currently it doesn't matter much because low_watermark is not used at all
> > as far as high_watermark is LLONG_MAX.
> >
>
> Don't we use by checking res_counter_below_low_watermark()?
```

yes, but only when we get above highwatermark.

YAMAMOTO Takashi

Containers mailing list
Containers@lists.linux-foundation.org

Subject: Re: [RFC] [PATCH] memory controller background reclamation

Posted by [Balbir Singh](#) on Mon, 26 Nov 2007 03:56:05 GMT

[View Forum Message](#) <> [Reply to Message](#)

Balbir Singh wrote:

> YAMAMOTO Takashi wrote:

>>>> + int batch_count = 128; /* XXX arbitrary */

>>> Could we define and use something like MEM_CGROUP_BATCH_COUNT for now?

>>> Later we could consider and see if it needs to be tunable. numbers are

>>> hard to read in code.

>> although i don't think it makes sense, i can do so if you prefer.

>>

>

> Using numbers like 128 make the code unreadable. I prefer something

> like MEM_CGROUP_BATCH_COUNT since its more readable than 128. If we ever

> propagate batch_count to other dependent functions, I'd much rather do

> it with a well defined name.

>

I just checked we already have FORCE_UNCHARGE_BATCH, we could just rename and re-use it.

--

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] [PATCH] memory controller background reclamation

Posted by [Balbir Singh](#) on Mon, 26 Nov 2007 04:09:04 GMT

[View Forum Message](#) <> [Reply to Message](#)

YAMAMOTO Takashi wrote:

>> I'll start some tests on these patches.

>

> thanks.

>

I tried testing, but the patch failed to apply. I think that's because it was ported on top of the NUMA patches by KAMEZAWA-San. I'll rebase and test.

--

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] [PATCH] memory controller background reclamation
Posted by [yamamoto](#) on Mon, 26 Nov 2007 22:43:58 GMT
[View Forum Message](#) <> [Reply to Message](#)

> Balbir Singh wrote:
> > YAMAMOTO Takashi wrote:
> >>> + int batch_count = 128; /* XXX arbitrary */
> >>> Could we define and use something like MEM_CGROUP_BATCH_COUNT for now?
> >>> Later we could consider and see if it needs to be tunable. numbers are
> >>> hard to read in code.
> >> although i don't think it makes sense, i can do so if you prefer.
> >>
> >
> > Using numbers like 128 make the code unreadable. I prefer something
> > like MEM_CGROUP_BATCH_COUNT since it's more readable than 128. If we ever
> > propagate batch_count to other dependent functions, I'd much rather do
> > it with a well defined name.
> >
> >
> I just checked we already have FORCE_UNCHARGE_BATCH, we could just
> rename and re-use it.

i don't think it's a good idea to use a single constant for completely different things.

YAMAMOTO Takashi

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

Subject: Re: [RFC] [PATCH] memory controller background reclamation

Posted by [yamamoto](#) on Mon, 26 Nov 2007 23:46:19 GMT

[View Forum Message](#) <> [Reply to Message](#)

> Using numbers like 128 make the code unreadable. I prefer something
> like MEM_CGROUP_BATCH_COUNT since its more readable than 128. If we ever
> propagate batch_count to other dependent functions, I'd much rather do
> it with a well defined name.

here's a new version.

changes from the previous:

- define a constant. (MEM_CGROUP_BG_RECLAIM_BATCH_COUNT)
no functional changes.
- don't increment ALLOCSTALL vm event for mem cgroup because
it isn't appropriate esp. for background reclamation.
introduce MEM_CGROUP_STAT_ALLOCSTALL instead.
(its value is same as memory.failcnt unless GFP_ATOMIC.)

YAMAMOTO Takashi

Signed-off-by: YAMAMOTO Takashi <yamamoto@valinux.co.jp>

```
---
--- linux-2.6.24-rc2-mm1-kame-pd/include/linux/res_counter.h.BACKUP 2007-11-14
16:05:48.000000000 +0900
+++ linux-2.6.24-rc2-mm1-kame-pd/include/linux/res_counter.h 2007-11-22 15:14:32.000000000
+0900
@@ -32,6 +32,13 @@ struct res_counter {
    * the number of unsuccessful attempts to consume the resource
    */
    unsigned long long failcnt;
+
+ /*
+  * watermarks
+  */
+    unsigned long long high_watermark;
+    unsigned long long low_watermark;
+
+ /*
+  * the lock to protect all of the above.
+  * the routines below consider this to be IRQ-safe
@@ -66,6 +73,8 @@ enum {
    RES_USAGE,
    RES_LIMIT,
    RES_FAILCNT,
+    RES_HIGH_WATERMARK,
+    RES_LOW_WATERMARK,
};
```

```

/*
@@ -124,4 +133,26 @@ static inline bool res_counter_check_und
    return ret;
}

+static inline bool res_counter_below_low_watermark(struct res_counter *cnt)
+{
+ bool ret;
+ unsigned long flags;
+
+ spin_lock_irqsave(&cnt->lock, flags);
+ ret = cnt->usage < cnt->low_watermark;
+ spin_unlock_irqrestore(&cnt->lock, flags);
+ return ret;
+}
+
+static inline bool res_counter_above_high_watermark(struct res_counter *cnt)
+{
+ bool ret;
+ unsigned long flags;
+
+ spin_lock_irqsave(&cnt->lock, flags);
+ ret = cnt->usage > cnt->high_watermark;
+ spin_unlock_irqrestore(&cnt->lock, flags);
+ return ret;
+}
+
#endif
--- linux-2.6.24-rc2-mm1-kame-pd/kernel/res_counter.c.BACKUP 2007-11-14
16:05:52.000000000 +0900
+++ linux-2.6.24-rc2-mm1-kame-pd/kernel/res_counter.c 2007-11-22 15:14:32.000000000 +0900
@@ -17,6 +17,8 @@ void res_counter_init(struct res_counter
{
    spin_lock_init(&counter->lock);
    counter->limit = (unsigned long long)LLONG_MAX;
+ counter->high_watermark = (unsigned long long)LLONG_MAX;
+ counter->low_watermark = (unsigned long long)LLONG_MAX;
}

int res_counter_charge_locked(struct res_counter *counter, unsigned long val)
@@ -69,6 +71,10 @@ res_counter_member(struct res_counter *c
    return &counter->limit;
    case RES_FAILCNT:
        return &counter->failcnt;
+ case RES_HIGH_WATERMARK:
+ return &counter->high_watermark;
+ case RES_LOW_WATERMARK:

```

```

+ return &counter->low_watermark;
};

BUG();
@@ -99,6 +105,7 @@ ssize_t res_counter_write(struct res_cou
int ret;
char *buf, *end;
unsigned long long tmp, *val;
+ unsigned long flags;

buf = kmalloc(nbytes + 1, GFP_KERNEL);
ret = -ENOMEM;
@@ -122,9 +129,29 @@ ssize_t res_counter_write(struct res_cou
goto out_free;
}

+ spin_lock_irqsave(&counter->lock, flags);
+ val = res_counter_member(counter, member);
+ /* ensure low_watermark <= high_watermark <= limit */
+ switch (member) {
+ case RES_LIMIT:
+ if (tmp < counter->high_watermark)
+ goto out_locked;
+ break;
+ case RES_HIGH_WATERMARK:
+ if (tmp > counter->limit || tmp < counter->low_watermark)
+ goto out_locked;
+ break;
+ case RES_LOW_WATERMARK:
+ if (tmp > counter->high_watermark)
+ goto out_locked;
+ break;
+ }
+ *val = tmp;
+ BUG_ON(counter->high_watermark > counter->limit);
+ BUG_ON(counter->low_watermark > counter->high_watermark);
ret = nbytes;
+out_locked:
+ spin_unlock_irqrestore(&counter->lock, flags);
out_free:
kfree(buf);
out:
--- linux-2.6.24-rc2-mm1-kame-pd/mm/vmscan.c.BACKUP 2007-11-20 13:11:09.000000000
+0900
+++ linux-2.6.24-rc2-mm1-kame-pd/mm/vmscan.c 2007-11-27 08:09:51.000000000 +0900
@@ -1333,7 +1333,6 @@ static unsigned long do_try_to_free_page
unsigned long lru_pages = 0;
int i;

```

```

- count_vm_event(ALLOCSTALL);
/*
 * mem_cgroup will not do shrink_slab.
 */
@@ -1432,6 +1431,7 @@ unsigned long try_to_free_pages(struct z
.isolate_pages = isolate_pages_global,
};

+ count_vm_event(ALLOCSTALL);
return do_try_to_free_pages(zones, gfp_mask, &sc);
}

--- linux-2.6.24-rc2-mm1-kame-pd/mm/memcontrol.c.BACKUP 2007-11-20 13:11:09.000000000
+0900
+++ linux-2.6.24-rc2-mm1-kame-pd/mm/memcontrol.c 2007-11-27 08:27:10.000000000 +0900
@@ -28,6 +28,7 @@
#include <linux/rcupdate.h>
#include <linux/swap.h>
#include <linux/spinlock.h>
+#include <linux/workqueue.h>
#include <linux/fs.h>
#include <linux/seq_file.h>

@@ -35,6 +36,7 @@

struct cgroup_subsys mem_cgroup_subsys;
static const int MEM_CGROUP_RECLAIM_RETRIES = 5;
+static const int MEM_CGROUP_BG_RECLAIM_BATCH_COUNT = 128; /* XXX arbitrary */

/*
 * Statistics for memory cgroup.
@@ -45,6 +47,7 @@ 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_ALLOCSTALL, /* allocation stalled due to memory.limit */

MEM_CGROUP_STAT_NSTATS,
};
@@ -138,6 +141,10 @@ struct mem_cgroup {
 * statistics.
 */
struct mem_cgroup_stat stat;
+ /*
+ * background reclamation.
+ */
+ struct work_struct reclaim_work;

```

```

};

/*
@@ -240,6 +247,21 @@ static unsigned long mem_cgroup_get_all_

static struct mem_cgroup init_mem_cgroup;

+static DEFINE_MUTEX(mem_cgroup_workqueue_init_lock);
+static struct workqueue_struct *mem_cgroup_workqueue;
+
+static void mem_cgroup_create_workqueue(void)
+{
+
+ if (mem_cgroup_workqueue != NULL)
+ return;
+
+ mutex_lock(&mem_cgroup_workqueue_init_lock);
+ if (mem_cgroup_workqueue == NULL)
+ mem_cgroup_workqueue = create_workqueue("mem_cgroup");
+ mutex_unlock(&mem_cgroup_workqueue_init_lock);
+}
+
static inline
struct mem_cgroup *mem_cgroup_from_cont(struct cgroup *cont)
{
@@ -566,6 +588,45 @@ unsigned long mem_cgroup_isolate_pages(u
return nr_taken;
}

+static void
+mem_cgroup_schedule_reclaim(struct mem_cgroup *mem)
+{
+
+ if (mem_cgroup_workqueue == NULL) {
+ BUG_ON(mem->css.cgroup->parent != NULL);
+ return;
+ }
+
+ if (work_pending(&mem->reclaim_work))
+ return;
+
+ css_get(&mem->css); /* XXX need some thoughts wrt cgroup removal. */
+ /*
+ * XXX workqueue is not an ideal mechanism for our purpose.
+ * revisit later.
+ */
+ if (!queue_work(mem_cgroup_workqueue, &mem->reclaim_work))
+ css_put(&mem->css);

```

```

+}
+
+static void
+mem_cgroup_reclaim(struct work_struct *work)
+{
+ struct mem_cgroup * const mem =
+ container_of(work, struct mem_cgroup, reclaim_work);
+ int batch_count = MEM_CGROUP_BG_RECLAIM_BATCH_COUNT;
+
+ for (; batch_count > 0; batch_count--) {
+ if (res_counter_below_low_watermark(&mem->res))
+ break;
+ if (!try_to_free_mem_cgroup_pages(mem, GFP_KERNEL))
+ break;
+ }
+ if (batch_count == 0)
+ mem_cgroup_schedule_reclaim(mem);
+ css_put(&mem->css);
+}
+
+/*
+ * Charge the memory controller for page usage.
+ * Return
@@ -631,6 +692,12 @@ retry:
    rcu_read_unlock();

    /*
+ * schedule background reclaim if we are above the high watermark.
+ */
+ if (res_counter_above_high_watermark(&mem->res))
+ mem_cgroup_schedule_reclaim(mem);
+
+ /*
+ * If we created the page_cgroup, we should free it on exceeding
+ * the cgroup limit.
+ */
@@ -642,6 +709,10 @@ retry:
    if (is_atomic)
        goto noreclaim;

+ preempt_disable();
+ __mem_cgroup_stat_add_safe(&mem->stat,
+ MEM_CGROUP_STAT_ALLOCSTALL, 1);
+ preempt_enable();
+ if (try_to_free_mem_cgroup_pages(mem, gfp_mask))
+ continue;

@@ -939,9 +1010,16 @@ static ssize_t mem_cgroup_write(struct c

```

```

    struct file *file, const char __user *userbuf,
    size_t nbytes, loff_t *ppos)
{
- return res_counter_write(&mem_cgroup_from_cont(cont)->res,
+ ssize_t ret;
+
+ ret = res_counter_write(&mem_cgroup_from_cont(cont)->res,
    cft->private, userbuf, nbytes, ppos,
    mem_cgroup_write_strategy);
+
+ if (ret >= 0 && cft->private == RES_HIGH_WATERMARK)
+ mem_cgroup_create_workqueue();
+
+ return ret;
}

static ssize_t mem_control_type_write(struct cgroup *cont,
@@ -1031,6 +1109,7 @@ static const struct mem_cgroup_stat_desc
} mem_cgroup_stat_desc[] = {
    [MEM_CGROUP_STAT_CACHE] = { "cache", PAGE_SIZE, },
    [MEM_CGROUP_STAT_RSS] = { "rss", PAGE_SIZE, },
+ [MEM_CGROUP_STAT_ALLOCSTALL] = { "allocstall", 1, },
};

static int mem_control_stat_show(struct seq_file *m, void *arg)
@@ -1097,6 +1176,18 @@ static struct cftype mem_cgroup_files[]
    .read = mem_cgroup_read,
    },
    {
+ .name = "high_watermark_in_bytes",
+ .private = RES_HIGH_WATERMARK,
+ .write = mem_cgroup_write,
+ .read = mem_cgroup_read,
+ },
+ {
+ .name = "low_watermark_in_bytes",
+ .private = RES_LOW_WATERMARK,
+ .write = mem_cgroup_write,
+ .read = mem_cgroup_read,
+ },
+ {
    .name = "control_type",
    .write = mem_control_type_write,
    .read = mem_control_type_read,
@@ -1161,6 +1252,8 @@ mem_cgroup_create(struct cgroup_subsys *
    if (alloc_mem_cgroup_per_zone_info(mem, node))
        goto free_out;

```

```
+ INIT_WORK(&mem->reclaim_work, mem_cgroup_reclaim);  
+  
  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>
