Subject: [RFC 1/2] memcg: hierarchy support core (yet another one)
Posted by KAMEZAWA Hiroyuki on Wed, 28 May 2008 07:56:20 GMT
View Forum Message <> Reply to Message

hierarchy support for memcg.

I know Balbir-san is now developping one. So this is yet-another-one.

Major difference is how to treat parent-child relationship. IIRC, his one allows
to share resource among parent and child. The wall between parent and child
was low. This one doesn't. The wall is high.
i.e. this one focuses on resource isolation, not on intelligent, rich controls.

Pros.
 - easy to use, easy to understand.
 - simple. low overheads.
Cons.
 - not provide rich controls, may not meet some middleware's requirements.

TODO?:
 - For allowing resource share between parent and child, maybe something
   rather than 'limit' is necessary..maybe idea like 'soft-limit' can work
   very well.

==
This patch tries to implements simple 'hierarchy policy' in res_counter.
Of couse, there is no influence to a user who doesn't want to handle hierarchy.

While several policy of hierarchy can be considered, this patch implements
simple one "the parent includes, overcommits the child". works as following.

 1. create a child. set default child limits to be 0.
 2. set limit to child.
    2-a. before setting limit to child, prepare enough room in parent.
    2-b. increase 'usage' of parent by child's limit.
 3. the child remembers what amount of resource is from its parent.
    the parent remembers what amout of resource is to the childs.

 Above means that
 - a directory's limit implies the sum of all subdirectories.
 - there are no shared resource between parent <-> child.

 Pros.
  - simple and hard and easy policy.
  - no heirarchy overhead.
  - no resource share among child <-> parent. very suiable for multilevel
    resource isolation.

```
 Cons.
  - not soft as to implement some kind of _intelligent_ hierarchy balancing
    in the kernel (but maybe middleware can do...)
  - no resource share among child <-> parent...seems not so wise ;)

TODO:
  - Find better words.....
  - For implementing  _intelligent_ hierarchy balancing, we'll have to
    add something...new parameter rather than limit ?

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

---
 Documentation/controllers/resource_counter.txt |   28 +++++
 include/linux/res_counter.h                    |   70 ++++++++++++++
 kernel/res_counter.c                           |  121 +++++++++++++++++++++++++--
 3 files changed, 211 insertions(+), 8 deletions(-)

Index: hie-2.6.26-rc2-mm1/include/linux/res_counter.h
===================================================================
--- hie-2.6.26-rc2-mm1.orig/include/linux/res_counter.h
+++ hie-2.6.26-rc2-mm1/include/linux/res_counter.h
@@ -39,6 +39,18 @@ struct res_counter {
    */
   unsigned long long failcnt;
   /*
+  * the amount of resource comes from parenet cgroup. Should be
+  * returned to the parent at destroying/resizing this res_counter.
+  */
+ unsigned long long borrow;
+ /*
+  * the sum of all resource which is borrowed by its child now.
+  * Limit can't be lower than this value. (i.e. resizing child before
+  * resizing parent.)
+  */
+ unsigned long long security;
+
+ /*
    * the lock to protect all of the above.
    * the routines below consider this to be IRQ-safe
    */
@@ -57,6 +69,7 @@ struct res_counter {
  * @nbytes:  its size...
  * @pos:     and the offset.
  */
+typedef int (*res_resize_callback_t)(struct res_counter *, unsigned long long);

 u64 res_counter_read_u64(struct res_counter *counter, int member);
```

```
@@ -65,8 +78,50 @@ ssize_t res_counter_read(struct res_coun
   int (*read_strategy)(unsigned long long val, char *s));
 ssize_t res_counter_write(struct res_counter *counter, int member,
   const char __user *buf, size_t nbytes, loff_t *pos,
-  int (*write_strategy)(char *buf, unsigned long long *val));
+  int (*write_strategy)(char *buf, unsigned long long *val),
+  res_resize_callback_t callback);
+
+/**
+ * Borrow resource from its parent. By this, res->usage of parent
+ * increased by 'val' and res->limit of child is set to 'val'.
+ * The child remembers how much it borrows resource from the parent in
+ * res->borrow member. The parent remembers how much resource is owned by
+ * its child in res->security member.
+ *
+ * @child:    an entity to set res->limit.
+ * @parent:   parent of child and source of resource.
+ * @val:      How much does child want to borrow from parent ?
+ * @callback: A callback for making resource to allow this borrowing, called
+ *            against parent. callback should returns 0 at success,
+ *       returns !0 at failure. _No_ lock is held while callback is
+ *       called. If NULL, no callback, no retry.
+ * @retry:    # of retries at calling callback for making resource.
+ *            -1 means inifite loop. At each retry, yield() is called.
+ * Returns 0 if success. !0 at failure.
+ *
+ */
+typedef int (*res_shrink_callback_t)(struct res_counter*, unsigned long long);

+int res_counter_borrow_resource(struct res_counter *child,
+    struct res_counter *parent,
+    unsigned long long val,
+    res_shrink_callback_t callback, int retry);
+
+/**
+ * Return resource to its parent.
+ * @child:   entry to resize. its limit will decreased by val.
+ * @parent:  resource will be re-added to this.
+ * @val  :   How much does child repay to parent ? -1 means 'all and force'.
+ * @callback: A callback for decreasing resouce usage of child before
+ *            repayment. If NULL, just deceases child's limit.
+ * @retry:   # of retries at calling callback for freeing resource.
+ *            -1 means inifite loop. At each retry, yield() is called.
+ * Returns 0 at success.
+ */
+int res_counter_repay_resource(struct res_counter *child,
+    struct res_counter *parent,
```

```
+    unsigned long long val,
+    res_shrink_callback_t callback, int retry);
 /*
  * the field descriptors. one for each member of res_counter
  */
@@ -76,6 +131,8 @@ enum {
  RES_MAX_USAGE,
  RES_LIMIT,
  RES_FAILCNT,
+ RES_BORROW,
+ RES_SECURITY,
 };

 /*
@@ -153,4 +210,15 @@ static inline void res_counter_reset_fai
  cnt->failcnt = 0;
  spin_unlock_irqrestore(&cnt->lock, flags);
 }
+
+/*
+ * should be called only after cgroup creation.
+ */
+static inline void res_counter_zero_limit(struct res_counter *cnt)
+{
+ unsigned long flags;
+ spin_lock_irqsave(&cnt->lock, flags);
+ cnt->limit = 0;
+ spin_unlock_irqrestore(&cnt->lock, flags);
+}
 #endif
Index: hie-2.6.26-rc2-mm1/kernel/res_counter.c
===================================================================
--- hie-2.6.26-rc2-mm1.orig/kernel/res_counter.c
+++ hie-2.6.26-rc2-mm1/kernel/res_counter.c
@@ -74,6 +74,8 @@ res_counter_member(struct res_counter *c
   return &counter->limit;
  case RES_FAILCNT:
   return &counter->failcnt;
+ case RES_SECURITY:
+  return &counter->security;
  };

  BUG();
@@ -104,7 +106,8 @@ u64 res_counter_read_u64(struct res_coun

 ssize_t res_counter_write(struct res_counter *counter, int member,
   const char __user *userbuf, size_t nbytes, loff_t *pos,
-  int (*write_strategy)(char *st_buf, unsigned long long *val))
```

```
+  int (*write_strategy)(char *st_buf, unsigned long long *val),
+  res_resize_callback_t callback)
 {
  int ret;
  char *buf, *end;
@@ -133,13 +136,119 @@ ssize_t res_counter_write(struct res_cou
   if (*end != '\0')
    goto out_free;
  }
- spin_lock_irqsave(&counter->lock, flags);
- val = res_counter_member(counter, member);
- *val = tmp;
- spin_unlock_irqrestore(&counter->lock, flags);
- ret = nbytes;
+ if (member != RES_LIMIT || !callback) {
+  spin_lock_irqsave(&counter->lock, flags);
+  val = res_counter_member(counter, member);
+  *val = tmp;
+  spin_unlock_irqrestore(&counter->lock, flags);
+  ret = nbytes;
+ } else {
+  /* call a callback for hierarchy management */
+  ret = callback(counter, tmp);
+  if (!ret)
+   ret = nbytes;
+ }
+
 out_free:
  kfree(buf);
 out:
  return ret;
 }
+
+/*
+ * Borrow resource from its parent to maintain hierarchy of cgroup.
+ * See res_counter.h for detail.
+ */
+
+int res_counter_borrow_resource(struct res_counter *child,
+    struct res_counter *parent,
+    unsigned long long val,
+    res_shrink_callback_t callback, int retry)
+{
+ int done = 0;
+ unsigned long flags;
+
+ /* Enough resources ? */
+ while (1) {
```

```
+  /* res_counter_charge just handles 'long' value...*/
+  spin_lock_irqsave(&parent->lock, flags);
+  if (parent->usage + val < parent->limit) {
+   parent->usage += val;
+   parent->security += val;
+   done = 1;
+  }
+  spin_unlock_irqrestore(&parent->lock, flags);
+  if (done)
+   break;
+  if (!retry || !callback)
+   goto fail;
+  if (retry > 0)
+   --retry;
+  yield();
+  callback(parent, val);
+ }
+
+ /* ok, we successfully got enough resource. */
+ spin_lock_irqsave(&child->lock, flags);
+ child->limit += val;
+ child->borrow += val;
+ spin_unlock_irqrestore(&child->lock, flags);
+
+ return 0;
+fail:
+ return 1;
+}
+
+/*
+ * Repay resource to its parent to maintain hierarchy of cgroup.
+ * See res_counter.h for detail.
+ */
+
+int res_counter_repay_resource(struct res_counter *child,
+    struct res_counter *parent,
+    unsigned long long val,
+    res_shrink_callback_t callback, int retry)
+{
+ unsigned long flags;
+ unsigned long long repay;
+ int done = 0;
+ /* Enough resources ? */
+ while (1) {
+  spin_lock_irqsave(&child->lock, flags);
+
+  if (val == (unsigned long long)-1) {
+   val = child->borrow;
```

```
+   child->limit = 0;
+   child->borrow = 0;
+   done = 1;
+  } else if (child->usage + val <= child->limit) {
+   child->limit -= val;
+   child->borrow -= val;
+   done = 1;
+  }
+  spin_unlock_irqrestore(&child->lock, flags);
+  if (done)
+   break;
+  if (!retry-- || !callback)
+   goto fail;
+  /*
+   * we want to rest somewhere but right after callback is
+   * not good place. So rest here.
+   */
+  yield();
+  /* reduce resource usage */
+  callback(child, val);
+ }
+
+ /* ok, we successfully got enough resource. */
+ spin_lock_irqsave(&parent->lock, flags);
+ BUG_ON(parent->security < val);
+ BUG_ON(parent->usage < val);
+ parent->security -= val;
+ parent->usage -= val;
+ spin_unlock_irqrestore(&parent->lock, flags);
+
+ return 0;
+fail:
+ return 1;
+}
Index: hie-2.6.26-rc2-mm1/Documentation/controllers/resource_counter.txt
===================================================================
--- hie-2.6.26-rc2-mm1.orig/Documentation/controllers/resource_counter.txt
+++ hie-2.6.26-rc2-mm1/Documentation/controllers/resource_counter.txt
@@ -39,10 +39,15 @@ to work with it.
   The failcnt stands for "failures counter". This is the number of
  resource allocation attempts that failed.

- c. spinlock_t lock
+ e. spinlock_t lock

   Protects changes of the above values.

+ f. borrow
```

+ The amount of resouce got from its parent.
+
+ g. security
+ The amount of resouce assigned to its childs.

2. Basic accounting routines
@@ -179,3 +184,24 @@ counter fields. They are recommended to
still can help with it).

c. Compile and run :)
+
+6. Hirarchy Model
+ 1) simple isolation hierarchy.
+ res_counter supports a simple hierarchy model as that the child's resouce
+ is borrowed from its parent.
+
+ When the limit is set to a child, its parent's usage increases by the
+ amount of limit. i.e. the child borrows resource from its parent when
+ it set the limit.
+
+ This kind of hierarhcy is very useful when you implements multilevel
+ hierarchy as multilevel resource isolation.
+ A) admin - user
+ - system admin layer ....the first level
+ - user layer ....the second level for user A, B, C
+ B) application/service layer.
+ - application layre ... the first level
+ - service layer ... the second level for service Gold, Silver,...
+
+ see res_counter_borrow_resource() and res_counter_repay_resouce().
+

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