

[RFC][PATCH 1/2] memcg: res_counter hierarchy

Source: <http://linux.derkeiler.com/Mailing-Lists/Kernel/2008-05/msg13612.html>

- *From:* KAMEZAWA Hiroyuki <kamezawa.hiroyu@xxxxxxxxxxxxxxxx>
 - *Date:* Fri, 30 May 2008 10:45:15 +0900
-

This patch tries to implements `_simple_` 'hierarchy policy' in `res_counter`.

While several policy of hierarchy can be considered, this patch implements simple one

- the parent includes, over-commits the child
- there are no shared resource
- dynamic hierarchy resource usage management in the kernel is not necessary

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 sets its limit to the val moved from the parent.
the parent remembers what amount of resource is to the children.

Above means that

- a directory's usage implies the sum of all sub directories + own usage.
- there are no shared resource between parent <-> child.

Pros.

- simple and easy policy.
- no hierarchy overhead.
- no resource share among child <-> parent. very suitable for multilevel resource isolation.

Cons.

- not good to implement some kind of `_intelligent_` hierarchy balancing in the `_kernel_`

Changelog:

- removed borrow.
- fixed tons of typos.

Signed-off-by: KAMEZAWA Hiroyuki <kamezawa.hiroyu@xxxxxxxxxxxxxxxx>

Documentation/controllers/resource_counter.txt | 28 +++++

[RFC][PATCH 1/2] memcg: res_counter hierarchy

include/linux/res_counter.h | 72 +++++
kernel/res_counter.c | 130 +++++
3 files changed, 222 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 +39,11 @@ struct res_counter {  
*/  
unsigned long long failcnt;  
/*  
+ * the sum of all resource which is assigned to children.  
+ */  
+ unsigned long long for_children;  
+  
+ /*  
+ * the lock to protect all of the above.  
+ * the routines below consider this to be IRQ-safe  
+ */  
@@ -57,6 +62,7 @@ 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 +71,55 @@ 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);  
+  
+/**  
+ * Move resource from parent to child and set child's limit.  
+ * By this,  
+ * res->usage of the parent increased by 'val'  
+ * res->for_children of the parent increased by 'val'  
+ * res->limit of the child increased by 'val'  
+ *  
+ * @child: an entity to set res->limit.  
+ * @parent: parent of child and source of resource.  
+ * @val: How much does child want to move from parent ?  
+ * @callback: A callback for making resource to allow this moving, called  
+ * against parent. callback should returns 0 at success,  
+ * returns !0 at failure. _No_lock is held while the callback is  
+ * called. If no callback(NULL), no retry.  
+ * @retry: # of retries at calling callback for making resource.
```

[RFC][PATCH 1/2] memcg: res_counter hierarchy

```
+ * -1 means infinite 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.
+ * By this,
+ * res->usage of the parent is decreased by 'val'
+ * res->for_children of the parent is decreased by 'val'
+ * res->limit of the child is decreased by 'val'
+ *
+ * @child: entry to resize.
+ * @parent: resource will be moved back to this.
+ * @val : How much does child repay to parent ? -1 means 'all'
+ * @callback: A callback for decreasing resource usage of child before
+ * moving. If NULL, just decreases child's limit.
+ * @retry: # of retries at calling callback for freeing resource.
+ * -1 means infinite 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 +129,7 @@ enum {
RES_MAX_USAGE,
RES_LIMIT,
RES_FAILCNT,
+ RES_FOR_CHILDREN,
};

+/*
@@ -104,6 +158,11 @@ enum charge_code __must_check res_counte

enum charge_code __must_check res_counter_charge(struct res_counter *counter,
unsigned long val);
+/*
+ * Compare usage with ->borrow member instead of ->limit.
+ */
+int __must_check res_counter_charge_borrow(struct res_counter *counter,
+ unsigned long val);
```

[RFC][PATCH 1/2] memcg: res_counter hierarchy

```
/*
 * uncharge – tell that some portion of the resource is released
 @@ -158,4 +217,15 @@ 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
@@ -76,6 +76,8 @@ res_counter_member(struct res_counter *c
return &counter->limit;
case RES_FAILCNT:
return &counter->failcnt;
+ case RES_FOR_CHILDREN:
+ return &counter->for_children;
};

BUG();
@@ -106,7 +108,8 @@ 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;
@@ -135,13 +138,118 @@ 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;
```

[RFC][PATCH 1/2] memcg: res_counter hierarchy

```
+ 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;
}
+
+/*
+ * This tries to move 'val' resource from parent. At success,
+ * child->limit += val.
+ * parent->for_children += val.
+ * parent->usage += val.
+ */
+
+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 success = 0;
+ unsigned long flags;
+
+ /* Borrow resource from parent */
+ success = 0;
+ 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->for_children += val;
+ success = 1;
+ }
+ spin_unlock_irqrestore(&parent->lock, flags);
+ if (success)
+ 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;
+ spin_unlock_irqrestore(&child->lock, flags);
+ return 0;
+fail:
+ return 1;
+ }
+
+ /*
+  * Move resource to its parent.
+  * child->limit -= val.
+  * parent->usage -= val.
+  * parent->limit -= val.
+  */
+
+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;
+ int done = 0;
+ /* Enough resources ? */
+ while (1) {
+ spin_lock_irqsave(&child->lock, flags);
+
+ if (val == (unsigned long long)-1) {
+ val = child->limit;
+ child->limit = 0;
+ done = 1;
+ } else if (child->usage + val <= child->limit) {
+ child->limit -= 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);
+ }

```

[RFC][PATCH 1/2] memcg: res_counter hierarchy

```
+
+ /* ok, we successfully got enough resource. */
+ spin_lock_irqsave(&parent->lock, flags);
+ BUG_ON(parent->for_children < val);
+ BUG_ON(parent->usage < val);
+ parent->for_children -= 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 +39,13 @@ 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. for_children
+ The amount of resource moved to the children.
+

2. Basic accounting routines

@@ -179,3 +182,24 @@ counter fields. They are recommended to still can help with it).

c. Compile and run :)

```
+
+6. Hierarchy Model
+ 1) simple isolation hierarchy.
+ res_counter supports a simple hierarchy model as that the child's resource
+ is moved 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.
+
+ hirarchical cgroup is very useful when you implements hierarchical
+ resource isolation. For example,
+ A) admin - user
+ - system admin layer ....the first level
+ - user layer ....the second level for user A, B, C
```

[RFC][PATCH 1/2] memcg: res_counter hierarchy

- + B) application/service layer.
- + – application layer ... the first level
- + – service layer ... the second level for service Gold, Silver,...
- +
- + see res_counter_borrow_resource() and res_counter_repay_resource().
- +

—

To unsubscribe from this list: send the line "unsubscribe linux-kernel" in the body of a message to majordomo@xxxxxxxxxxxxxxxxx

More majordomo info at <http://vger.kernel.org/majordomo-info.html>

Please read the FAQ at <http://www.tux.org/lkml/>