Subject: Re: [ckrm-tech] [RFC][PATCH 2/7] UBC: core (structures, API)
Posted by dev on Fri, 18 Aug 2006 11:34:22 GMT
View Forum Message <> Reply to Message

Matt Helsley wrote:

[snip]
>>+obj-$(CONFIG_USER_RESOURCE) += beancounter.o
>>--- /dev/null 2006-07-18 14:52:43.075228448 +0400
>>+++ ./kernel/ub/beancounter.c 2006-08-10 15:09:34.000000000 +0400
>>@@ -0,0 +1,398 @@
>>+/*
>>+ * kernel/ub/beancounter.c
>>+ *
>>+ * Copyright (C) 2006 OpenVZ. SWsoft Inc
>>+ * Original code by (C) 1998   Alan Cox
>>+ *   1998-2000 Andrey Savochkin <saw@saw.sw.com.sg>
>>+ */
>>+
>>+#include <linux/slab.h>
>>+#include <linux/module.h>
>>+
>>+#include <ub/beancounter.h>
>>+
>>+static kmem_cache_t *ub_cachep;
>>+static struct user_beancounter default_beancounter;
>>+static struct user_beancounter default_subbeancounter;
>>+
>>+static void init_beancounter_struct(struct user_beancounter *ub, uid_t id);
>>+
>>+struct user_beancounter ub0;
>>+
>>+const char *ub_rnames[] = {
>>+};
>>+
>>+#define ub_hash_fun(x) ((((x) >> 8) ^ (x)) & (UB_HASH_SIZE - 1))
>>+#define ub_subhash_fun(p, id) ub_hash_fun((p)->ub_uid + (id) * 17)
>>+
>>+struct hlist_head ub_hash[UB_HASH_SIZE];
>>+spinlock_t ub_hash_lock;
>>+
>>+EXPORT_SYMBOL(ub_hash);
>>+EXPORT_SYMBOL(ub_hash_lock);
>>+
>>+/*
>>+ * Per user resource beancounting. Resources are tied to their luid.
>
>

> You haven't explained what an luid is at this point in the patch series.
> Patch 0 says:
>
> diff-ubc-syscalls.patch:
>   Patch adds system calls for UB management:
>   1. sys_getluid   - get current UB id
>
> But I have no idea what that l is there for. Why not sys_get_ubid() for
> instance?
it is a historical name given by Alan Cox (imho) and Andrey Savochkin.
will rename it to sys_getubid, sys_setubid and field
ub_uid will be renamed into ub_id.

>>+ * The resource structure itself is tagged both to the process and
>>+ * the charging resources (a socket doesn't want to have to search for
>>+ * things at irq time for example). Reference counters keep things in
>>+ * hand.
>>+ *
>>+ * The case where a user creates resource, kills all his processes and
>>+ * then starts new ones is correctly handled this way. The refcounters
>>+ * will mean the old entry is still around with resource tied to it.
>>+ */
>>+
>
>
> So we create one beancounter object for every resource the user's tasks
> allocate? For instance, one per socket? Or does "resource structure"
> refer to something else?
no, user_beancounter structure exists one for a set of processes and its resources.

>>+struct user_beancounter *beancounter_findcreate(uid_t uid,
>>+ struct user_beancounter *p, int mask)
>>+{
>>+ struct user_beancounter *new_ub, *ub, *tmpl_ub;
>>+ unsigned long flags;
>>+ struct hlist_head *slot;
>>+ struct hlist_node *pos;
>>+
>>+ if (mask & UB_LOOKUP_SUB) {
>>+ WARN_ON(p == NULL);
>>+ tmpl_ub = &default_subbeancounter;
>>+ slot = &ub_hash[ub_subhash_fun(p, uid)];
>>+ } else {
>>+ WARN_ON(p != NULL);
>>+ tmpl_ub = &default_beancounter;
>>+ slot = &ub_hash[ub_hash_fun(uid)];
>>+ }
>>+ new_ub = NULL;

```
>>+
>>+retry:
>>+ spin_lock_irqsave(&ub_hash_lock, flags);
>>+ hlist_for_each_entry (ub, pos, slot, hash)
>>+  if (ub->ub_uid == uid && ub->parent == p)
>>+   break;
>>+
>>+ if (pos != NULL) {
>>+  get_beancounter(ub);
>>+  spin_unlock_irqrestore(&ub_hash_lock, flags);
>>+
>>+  if (new_ub != NULL) {
>>+   put_beancounter(new_ub->parent);
>>+   kmem_cache_free(ub_cachep, new_ub);
>>+  }
>>+  return ub;
>>+ }
>>+
>>+ if (!(mask & UB_ALLOC))
>>+  goto out_unlock;
>>+
>>+ if (new_ub != NULL)
>>+  goto out_install;
>>+
>>+ if (mask & UB_ALLOC_ATOMIC) {
>
>
> This block..
>
>
>>+  new_ub = kmem_cache_alloc(ub_cachep, GFP_ATOMIC);
>>+  if (new_ub == NULL)
>>+   goto out_unlock;
>>+
>>+  memcpy(new_ub, tmpl_ub, sizeof(*new_ub));
>>+  init_beancounter_struct(new_ub, uid);
>>+  if (p)
>>+   new_ub->parent = get_beancounter(p);
>
>
> ending here is almost exactly the same as the block ..
>
>
>>+  goto out_install;
>>+ }
>>+
>>+ spin_unlock_irqrestore(&ub_hash_lock, flags);
>>+
```

>
>
>>From here..
>
>
>>+ new_ub = kmem_cache_alloc(ub_cachep, GFP_KERNEL);
>>+ if (new_ub == NULL)
>>+ goto out;
>>+
>>+ memcpy(new_ub, tmpl_ub, sizeof(*new_ub));
>>+ init_beancounter_struct(new_ub, uid);
>>+ if (p)
>>+ new_ub->parent = get_beancounter(p);
>
>
> to here. You could make a flag variable that holds GFP_ATOMIC or
> GFP_KERNEL based on mask & UB_ALLOC_ATOMIC and perhaps turn this block
> into a small helper.
yeah, Oleg Nesterov already pointed to this. will cleanup.

>>+ goto retry;
>>+
>>+out_install:
>>+ hlist_add_head(&new_ub->hash, slot);
>>+out_unlock:
>>+ spin_unlock_irqrestore(&ub_hash_lock, flags);
>>+out:
>>+ return new_ub;
>>+}
>>+
>>+EXPORT_SYMBOL(beancounter_findcreate);
>>+
>>+void ub_print_uid(struct user_beancounter *ub, char *str, int size)
>>+{
>>+ if (ub->parent != NULL)
>>+ snprintf(str, size, "%u.%u", ub->parent->ub_uid, ub->ub_uid);
>>+ else
>>+ snprintf(str, size, "%u", ub->ub_uid);
>>+}
>>+
>>+EXPORT_SYMBOL(ub_print_uid);
>
>
>>From what I can see this patch doesn't really justify the need for the
> EXPORT_SYMBOL. Shouldn't that be done in the patch where it's needed
> outside of the kernel/ub code itself?
AFAIK these exports were used in our checkpointing code (OpenVZ)
ok, will remove exports from this patch series...

```
>>+void ub_print_resource_warning(struct user_beancounter *ub, int res,
>>+  char *str, unsigned long val, unsigned long held)
>>+{
>>+ char uid[64];
>>+
>>+ ub_print_uid(ub, uid, sizeof(uid));
>>+ printk(KERN_WARNING "UB %s %s warning: %s "
>>+   "(held %lu, fails %lu, val %lu)\n",
>>+   uid, ub_rnames[res], str,
>>+   (res < UB_RESOURCES ? ub->ub_parms[res].held : held),
>>+   (res < UB_RESOURCES ? ub->ub_parms[res].failcnt : 0),
>>+   val);
>>+}
>>+
>>+EXPORT_SYMBOL(ub_print_resource_warning);
>>+
>>+static inline void verify_held(struct user_beancounter *ub)
>>+{
>>+ int i;
>>+
>>+ for (i = 0; i < UB_RESOURCES; i++)
>>+  if (ub->ub_parms[i].held != 0)
>>+   ub_print_resource_warning(ub, i,
>>+     "resource is held on put", 0, 0);
>>+}
>>+
>>+void __put_beancounter(struct user_beancounter *ub)
>>+{
>>+ unsigned long flags;
>>+ struct user_beancounter *parent;
>>+
>>+again:
>>+ parent = ub->parent;
>>+ /* equevalent to atomic_dec_and_lock_irqsave() */
>
>
> nit: s/que/qui/
thanks!

>>+ local_irq_save(flags);
>>+ if (likely(!atomic_dec_and_lock(&ub->ub_refcount, &ub_hash_lock))) {
>>+  if (unlikely(atomic_read(&ub->ub_refcount) < 0))
>>+   printk(KERN_ERR "UB: Bad ub refcount: ub=%p, "
>>+     "luid=%d, ref=%d\n",
>>+     ub, ub->ub_uid,
>>+     atomic_read(&ub->ub_refcount));
>
```

>
> This seems to be for debugging purposes only. If not, perhaps this
> printk ought to be rate limited?
this is BUG_ON with more description. there is no much need to ratelimit it
as when it happens something really wrong happened in your system.

>>+ local_irq_restore(flags);
>>+ return;
>>+ }
>>+
>>+ if (unlikely(ub == &ub0)) {
>>+ printk(KERN_ERR "Trying to put ub0\n");
>
>
> Same thing for this printk.
will replace it with real BUG_ON here.

>>+ spin_unlock_irqrestore(&ub_hash_lock, flags);
>>+ return;
>>+ }
>>+
>>+ verify_held(ub);
>>+ hlist_del(&ub->hash);
>>+ spin_unlock_irqrestore(&ub_hash_lock, flags);
>>+
>>+ kmem_cache_free(ub_cachep, ub);
>>+
>>+ ub = parent;
>>+ if (ub != NULL)
>>+ goto again;
>
>
> Couldn't this be replaced by a do { } while (ub != NULL); loop?
this is ugly from indentation POV. also restarts are frequently used everywhere...

>>+}
>>+
>>+EXPORT_SYMBOL(__put_beancounter);
>>+
>>+/*
>>+ * Generic resource charging stuff
>>+ */
>>+
>>+int __charge_beancounter_locked(struct user_beancounter *ub,
>>+ int resource, unsigned long val, enum severity strict)
>>+{
>>+ /*
>>+ * ub_value <= UB_MAXVALUE, value <= UB_MAXVALUE, and only one addition

```
>>+  * at the moment is possible so an overflow is impossible.
>>+  */
>>+ ub->ub_parms[resource].held += val;
>>+
>>+ switch (strict) {
>>+  case UB_BARRIER:
>>+   if (ub->ub_parms[resource].held >
>>+     ub->ub_parms[resource].barrier)
>>+    break;
>>+   /* fallthrough */
>>+  case UB_LIMIT:
>>+   if (ub->ub_parms[resource].held >
>>+     ub->ub_parms[resource].limit)
>>+    break;
>>+   /* fallthrough */
>>+  case UB_FORCE:
>>+   ub_adjust_held_minmax(ub, resource);
>>+   return 0;
>>+  default:
>>+   BUG();
>>+ }
>>+
>>+ ub->ub_parms[resource].failcnt++;
>>+ ub->ub_parms[resource].held -= val;
>>+ return -ENOMEM;
>>+}
>>+
>>+int charge_beancounter(struct user_beancounter *ub,
>>+  int resource, unsigned long val, enum severity strict)
>>+{
>>+ int retval;
>>+ struct user_beancounter *p, *q;
>>+ unsigned long flags;
>>+
>>+ retval = -EINVAL;
>>+ BUG_ON(val > UB_MAXVALUE);
>>+
>>+ local_irq_save(flags);
>
>
> <factor>
>
>>+ for (p = ub; p != NULL; p = p->parent) {
>
>
> Seems rather expensive to walk up the tree for every charge. Especially
> if the administrator wants a fine degree of resource control and makes a
> tall tree. This would be a problem especially when it comes to resources
```

> that require frequent and fast allocation.
in heirarchical accounting you always have to update all the nodes :/
with flat UBC this doesn't introduce significant overhead.

>>+ spin_lock(&p->ub_lock);
>>+ retval = __charge_beancounter_locked(p, resource, val, strict);
>>+ spin_unlock(&p->ub_lock);
>>+ if (retval)
>>+   goto unroll;
>
>
> This can be factored by passing a flag that breaks the loop on an error:
>
>   if (retval && do_break_err)
>   return retval;
how about uncharge here?
didn't get your idea, sorry...

>
>
>>+ }
>
>
> </factor>
>
>>+out_restore:
>>+ local_irq_restore(flags);
>>+ return retval;
>>+
>
>
> <factor>
>
>>+unroll:
>>+ for (q = ub; q != p; q = q->parent) {
>>+ spin_lock(&q->ub_lock);
>>+ __uncharge_beancounter_locked(q, resource, val);
>>+ spin_unlock(&q->ub_lock);
>>+ }
>
>
> </factor>
>
>>+ goto out_restore;
>>+}
>>+
>>+EXPORT_SYMBOL(charge_beancounter);

>>+
>>+void charge_beancounter_notop(struct user_beancounter *ub,
>>+ int resource, unsigned long val)
>>+{
>>+ struct user_beancounter *p;
>>+ unsigned long flags;
>>+
>>+ local_irq_save(flags);
>
>
> <factor>
>
>>+ for (p = ub; p->parent != NULL; p = p->parent) {
>>+ spin_lock(&p->ub_lock);
>>+ __charge_beancounter_locked(p, resource, val, UB_FORCE);
>>+ spin_unlock(&p->ub_lock);
>>+ }
>
>
> <factor>
>
>>+ local_irq_restore(flags);
>
>
> Again, this could be factored with charge_beancounter using a helper
> function.
>
>
>>+}
>>+
>>+EXPORT_SYMBOL(charge_beancounter_notop);
>>+
>>+void __uncharge_beancounter_locked(struct user_beancounter *ub,
>>+ int resource, unsigned long val)
>>+{
>>+ if (unlikely(ub->ub_parms[resource].held < val)) {
>>+ ub_print_resource_warning(ub, resource,
>>+   "uncharging too much", val, 0);
>>+ val = ub->ub_parms[resource].held;
>>+ }
>>+ ub->ub_parms[resource].held -= val;
>>+ ub_adjust_held_minmax(ub, resource);
>>+}
>>+
>>+void uncharge_beancounter(struct user_beancounter *ub,
>>+ int resource, unsigned long val)
>>+{
>>+ unsigned long flags;

>>+ struct user_beancounter *p;
>>+
>>+ for (p = ub; p != NULL; p = p->parent) {
>>+ spin_lock_irqsave(&p->ub_lock, flags);
>>+ __uncharge_beancounter_locked(p, resource, val);
>>+ spin_unlock_irqrestore(&p->ub_lock, flags);
>>+ }
>>+}
>
>
> Looks like your unroll: label in charge_beancounter above.
ok, will introduce helpers.
>
>
>>+
>>+EXPORT_SYMBOL(uncharge_beancounter);
>>+
>>+void uncharge_beancounter_notop(struct user_beancounter *ub,
>>+ int resource, unsigned long val)
>>+{
>>+ struct user_beancounter *p;
>>+ unsigned long flags;
>>+
>>+ local_irq_save(flags);
>
>
> <factor>
>
>>+ for (p = ub; p->parent != NULL; p = p->parent) {
>>+ spin_lock(&p->ub_lock);
>>+ __uncharge_beancounter_locked(p, resource, val);
>>+ spin_unlock(&p->ub_lock);
>>+ }
>
>
> </factor>
>
>>+ local_irq_restore(flags);
>>+}
>>+
>>+EXPORT_SYMBOL(uncharge_beancounter_notop);
>>+
>>+/*
>>+ * Initialization
>>+ *
>>+ * struct user_beancounter contains
>>+ * - limits and other configuration settings
>>+ * - structural fields: lists, spinlocks and so on.

```
>>+ *
>>+ * Before these parts are initialized, the structure should be memset
>>+ * to 0 or copied from a known clean structure.  That takes care of a lot
>>+ * of fields not initialized explicitly.
>>+ */
>>+
>>+static void init_beancounter_struct(struct user_beancounter *ub, uid_t id)
>>+{
>>+ atomic_set(&ub->ub_refcount, 1);
>>+ spin_lock_init(&ub->ub_lock);
>>+ ub->ub_uid = id;
>>+}
>>+
>>+static void init_beancounter_nolimits(struct user_beancounter *ub)
>>+{
>>+ int k;
>>+
>>+ for (k = 0; k < UB_RESOURCES; k++) {
>>+  ub->ub_parms[k].limit = UB_MAXVALUE;
>>+  ub->ub_parms[k].barrier = UB_MAXVALUE;
>>+ }
>>+}
>>+
>>+static void init_beancounter_syslimits(struct user_beancounter *ub)
>>+{
>>+ int k;
>>+
>>+ for (k = 0; k < UB_RESOURCES; k++)
>>+  ub->ub_parms[k].barrier = ub->ub_parms[k].limit;
>>+}
>>+
>>+void __init ub_init_early(void)
>>+{
>>+ struct user_beancounter *ub;
>>+ struct hlist_head *slot;
>>+
>>+ ub = &ub0;
>>+
>
>
> <factor>
>
>>+ memset(ub, 0, sizeof(*ub));
>>+ init_beancounter_nolimits(ub);
>>+ init_beancounter_struct(ub, 0);
>>+
>
>
```

> </factor>
???
>
>>+ spin_lock_init(&ub_hash_lock);
>>+ slot = &ub_hash[ub_hash_fun(ub->ub_uid)];
>>+ hlist_add_head(&ub->hash, slot);
>>+}
>>+
>>+void __init ub_init_late(void)
>>+{
>>+ struct user_beancounter *ub;
>>+
>>+ ub_cachep = kmem_cache_create("user_beancounters",
>>+   sizeof(struct user_beancounter),
>>+   0, SLAB_HWCACHE_ALIGN, NULL, NULL);
>>+ if (ub_cachep == NULL)
>>+ panic("Can't create ubc caches\n");
>>+
>>+ ub = &default_beancounter;
>
>
> <factor>
>
>>+ memset(ub, 0, sizeof(default_beancounter));
>>+ init_beancounter_syslimits(ub);
<<<< this one is different from the above :)
>>+ init_beancounter_struct(ub, 0);
>>+
>
>
> </factor>
>
>>+ ub = &default_subbeancounter;
>
>
> <factor>
>
>>+ memset(ub, 0, sizeof(default_subbeancounter));
>>+ init_beancounter_nolimits(ub);
>>+ init_beancounter_struct(ub, 0);
>
>
> </factor>
>
>>+}
>
>
> Cheers,

> -Matt Helsley
>
>