
Subject: Re: [ckrm-tech] [RFC][PATCH 2/7] UBC: core (structures, API)

Posted by [Matt Helsley](#) on Fri, 18 Aug 2006 01:58:48 GMT

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

On Wed, 2006-08-16 at 19:37 +0400, Kirill Korotaev wrote:

> Core functionality and interfaces of UBC:

> find/create beancounter, initialization,

> charge/uncharge of resource, core objects' declarations.

>

> Basic structures:

> ubparm - resource description

> user_beancounter - set of resources, id, lock

>

> Signed-Off-By: Pavel Emelianov <xemul@sw.ru>

> Signed-Off-By: Kirill Korotaev <dev@sw.ru>

>

> ---

> include/ub/beancounter.h | 157 ++++++

> init/main.c | 4

> kernel/Makefile | 1

> kernel/ub/Makefile | 7

> kernel/ub/beancounter.c | 398 ++++++

> 5 files changed, 567 insertions(+)

>

> --- /dev/null 2006-07-18 14:52:43.075228448 +0400

> +++ ./include/ub/beancounter.h 2006-08-10 14:58:27.000000000 +0400

> @@ -0,0 +1,157 @@

> +/*

> + * include/ub/beancounter.h

> + *

> + * Copyright (C) 2006 OpenVZ. SWsoft Inc

> + *

> + */

> +

> + #ifndef _LINUX_BEANCOUNTER_H

> + #define _LINUX_BEANCOUNTER_H

> +

> + /*

> + * Resource list.

> + */

> +

> + #define UB_RESOURCES 0

> +

> + struct ubparm {

> + /*

> + * A barrier over which resource allocations are failed gracefully.

> + * e.g. if the amount of consumed memory is over the barrier further

> + * sbrk() or mmap() calls fail, the existing processes are not killed.

```

> + */
> + unsigned long barrier;
> + /* hard resource limit */
> + unsigned long limit;
> + /* consumed resources */
> + unsigned long held;
> + /* maximum amount of consumed resources through the last period */
> + unsigned long maxheld;
> + /* minimum amount of consumed resources through the last period */
> + unsigned long minheld;
> + /* count of failed charges */
> + unsigned long failcnt;
> +};
> +
> +/*
> + * Kernel internal part.
> + */
> +
> +#ifdef __KERNEL__
> +
> +#include <linux/config.h>
> +#include <linux/spinlock.h>
> +#include <linux/list.h>
> +#include <asm/atomic.h>
> +
> +/*
> + * UB_MAXVALUE is essentially LONG_MAX declared in a cross-compiling safe form.
> + */
> +#define UB_MAXVALUE ( (1UL << (sizeof(unsigned long)*8-1)) - 1)
> +
> +
> +/*
> + * Resource management structures
> + * Serialization issues:
> + *  beancounter list management is protected via ub_hash_lock
> + *  task pointers are set only for current task and only once
> + *  refcount is managed atomically
> + *  value and limit comparison and change are protected by per-ub spinlock
> + */
> +
> +struct user_beancounter
> +{
> + atomic_t ub_refcount;
> + spinlock_t ub_lock;
> + uid_t ub_uid;
> + struct hlist_node hash;
> +
> + struct user_beancounter *parent;

```

```

> + void *private_data;
> +
> + /* resources statistics and settings */
> + struct ubparm ub_parms[UB_RESOURCES];
> +};
> +
> +enum severity { UB_BARRIER, UB_LIMIT, UB_FORCE };
> +
> +/* Flags passed to beancounter_findcreate() */
> +#define UB_LOOKUP_SUB 0x01 /* Lookup subbeancounter */
> +#define UB_ALLOC 0x02 /* May allocate new one */
> +#define UB_ALLOC_ATOMIC 0x04 /* Allocate with GFP_ATOMIC */
> +
> +#define UB_HASH_SIZE 256
> +
> +#ifdef CONFIG_USER_RESOURCE
> +extern struct hlist_head ub_hash[];
> +extern spinlock_t ub_hash_lock;
> +
> +static inline void ub_adjust_held_minmax(struct user_beancounter *ub,
> + int resource)
> +{
> + if (ub->ub_parms[resource].maxheld < ub->ub_parms[resource].held)
> + ub->ub_parms[resource].maxheld = ub->ub_parms[resource].held;
> + if (ub->ub_parms[resource].minheld > ub->ub_parms[resource].held)
> + ub->ub_parms[resource].minheld = ub->ub_parms[resource].held;
> +}
> +
> +void ub_print_resource_warning(struct user_beancounter *ub, int res,
> + char *str, unsigned long val, unsigned long held);
> +void ub_print_uid(struct user_beancounter *ub, char *str, int size);
> +
> +int __charge_beancounter_locked(struct user_beancounter *ub,
> + int resource, unsigned long val, enum severity strict);
> +void charge_beancounter_notop(struct user_beancounter *ub,
> + int resource, unsigned long val);
> +int charge_beancounter(struct user_beancounter *ub,
> + int resource, unsigned long val, enum severity strict);
> +
> +void __uncharge_beancounter_locked(struct user_beancounter *ub,
> + int resource, unsigned long val);
> +void uncharge_beancounter_notop(struct user_beancounter *ub,
> + int resource, unsigned long val);
> +void uncharge_beancounter(struct user_beancounter *ub,
> + int resource, unsigned long val);
> +
> +struct user_beancounter *beancounter_findcreate(uid_t uid,
> + struct user_beancounter *parent, int flags);

```

```

> +
> +static inline struct user_beancounter *get_beancounter(
> + struct user_beancounter *ub)
> +{
> + atomic_inc(&ub->ub_refcount);
> + return ub;
> +}
> +
> +void __put_beancounter(struct user_beancounter *ub);
> +static inline void put_beancounter(struct user_beancounter *ub)
> +{
> + __put_beancounter(ub);
> +}
> +
> +void ub_init_early(void);
> +void ub_init_late(void);
> +void ub_init_proc(void);
> +
> +extern struct user_beancounter ub0;
> +extern const char *ub_rnames[];
> +
> +#else /* CONFIG_USER_RESOURCE */
> +
> +#define beancounter_findcreate(id, p, f) (NULL)
> +#define get_beancounter(ub) (NULL)
> +#define put_beancounter(ub) do { } while (0)
> +#define __charge_beancounter_locked(ub, r, v, s) (0)
> +#define charge_beancounter(ub, r, v, s) (0)
> +#define charge_beancounter_notop(ub, r, v) do { } while (0)
> +#define __uncharge_beancounter_locked(ub, r, v) do { } while (0)
> +#define uncharge_beancounter(ub, r, v) do { } while (0)
> +#define uncharge_beancounter_notop(ub, r, v) do { } while (0)
> +#define ub_init_early() do { } while (0)
> +#define ub_init_late() do { } while (0)
> +#define ub_init_proc() do { } while (0)
> +
> +#endif /* CONFIG_USER_RESOURCE */
> +#endif /* __KERNEL__ */
> +
> +#endif /* _LINUX_BEANCOUNTER_H */
> --- ./init/main.c.ubcore 2006-08-10 14:55:47.000000000 +0400
> +++ ./init/main.c 2006-08-10 14:57:01.000000000 +0400
> @@ -52,6 +52,8 @@
> #include <linux/debug_locks.h>
> #include <linux/lockdep.h>
>
> +#include <ub/beancounter.h>
> +

```

```

> #include <asm/io.h>
> #include <asm/bugs.h>
> #include <asm/setup.h>
> @@ -470,6 +472,7 @@ asmlinkage void __init start_kernel(void
>  early_boot_irqs_off();
>  early_init_irq_lock_class();
>
> + ub_init_early();
> /*
>  * Interrupts are still disabled. Do necessary setups, then
>  * enable them
> @@ -563,6 +566,7 @@ asmlinkage void __init start_kernel(void
> #endif
>  fork_init(num_physpages);
>  proc_caches_init();
> + ub_init_late();
>  buffer_init();
>  unnamed_dev_init();
>  key_init();
> --- ./kernel/Makefile.ubcore 2006-08-10 14:55:47.000000000 +0400
> +++ ./kernel/Makefile 2006-08-10 14:57:01.000000000 +0400
> @@ -12,6 +12,7 @@ obj-y    = sched.o fork.o exec_domain.o
>
>  obj-$(CONFIG_STACKTRACE) += stacktrace.o
>  obj-y += time/
> +obj-y += ub/
>  obj-$(CONFIG_DEBUG_MUTEXES) += mutex-debug.o
>  obj-$(CONFIG_LOCKDEP) += lockdep.o
>  ifeq ($(CONFIG_PROC_FS),y)
> --- /dev/null 2006-07-18 14:52:43.075228448 +0400
> +++ ./kernel/ub/Makefile 2006-08-10 14:57:01.000000000 +0400
> @@ -0,0 +1,7 @@
> +#
> +# User resources part (UBC)
> +#
> +# Copyright (C) 2006 OpenVZ. SWsoft Inc
> +#
> +
> +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_cache;
> + 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?

```

> + * 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?

```
> + 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.

```
> + 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?

```

> +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;

```

```
> + /* equivalent to atomic_dec_and_lock_irqsave() */
```

```
nit: s/que/qui/
```

```
> + 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?

```
> + local_irq_restore(flags);
> + return;
> + }
> +
> + if (unlikely(ub == &ub0)) {
> +   printk(KERN_ERR "Trying to put ub0\n");
```

Same thing for this printk.

```
> + 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?

```
> +}
> +
> +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.

```
> + 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;

> + }

</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.

```
> +
```

```

> +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);  
> + 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
