Subject: [RFC][PATCH 5/7] VPIDs: vpid/pid conversion in VPID enabled case
Posted by Kirill Korotaev on Thu, 02 Feb 2006 16:30:16 GMT
View Forum Message <> Reply to Message

This is the main patch which contains all vpid-to-pid conversions
and auxilliary stuff. Virtual pids are distinguished from real ones
by the VPID_BIT bit set. Conversion from vpid to pid and vice versa
is performed in two ways: fast way, when vpid and it's according pid
differ only in VPID_BIT bit set ("linear" case), and more complex way,
when pid may correspond to any vpid ("sparse" case) - in this case we
use a hash-table based mapping.

Note that this patch implies that we have a public vps_info_t pointer
type to represent VPS (otherwise it is useless) so that it can be used
in any virtualisation solution. Virtualization solution should have the
following "interface":
1. vps_info_t has member int id;
2. vps_info_t has member struct task_struct *init_task;
3. the following macros/functions are defined:
a. inside_vps() - returns true if current task is now
inside VPS;
b. task_inside_vps(task_t *) - returns true if task
belongs to VPS;
c. current_vps() - returns vps_info_t for current VPS;
d. task_vps(task_t *) - returns vps_info_t that task
belongs to;
e. set_sparce_vpid(vps_info_t) - switches VPS into state
when "sparce" conversion is used;
f. sparse_vpid(vps_info_t) - returns true if vps is in
"sparce" state and false if it is in "linear";
g. get_vps_tasks_num(vps_info_t) - returns the number of
tasks that belong to VPS.

Kirill

```
--- ./include/linux/pid.h.vpid_virt 2006-02-02 14:32:41.162807472 +0300
+++ ./include/linux/pid.h 2006-02-02 14:33:17.963212960 +0300
@@ -10,11 +10,17 @@ enum pid_type
  PIDTYPE_MAX
 };

+#define VPID_BIT 10
+#define VPID_DIV (1 << VPID_BIT)
+
 struct pid
 {
  /* Try to keep pid_chain in the same cacheline as nr for find_pid */
  int nr;
```

```
  struct hlist_node pid_chain;
+#ifdef CONFIG_VIRTUAL_PIDS
+ int vnr;
+#endif
  /* list of pids with the same nr, only one of them is in the hash */
  struct list_head pid_list;
 };
@@ -30,6 +36,52 @@ struct pid
 #define comb_pid_to_vpid(pid)  (pid)
 #define alloc_vpid(pid, vpid)  (pid)
 #define free_vpid(vpid)   do { } while (0)
+#else /* CONFIG_VIRTUAL_PIDS */
+#define __is_virtual_pid(pid)  ((pid) & VPID_DIV)
+#define is_virtual_pid(pid)  (__is_virtual_pid(pid) || \
+  (((pid) == 1) && inside_vps()))
+
+extern int vpid_to_pid(int pid);
+extern int __vpid_to_pid(int pid);
+extern pid_t pid_type_to_vpid(int type, pid_t pid);
+extern pid_t __pid_type_to_vpid(int type, pid_t pid);
+
+static inline int comb_vpid_to_pid(int vpid)
+{
+ int pid = vpid;
+
+ if (vpid > 0) {
+  pid = vpid_to_pid(vpid);
+  if (unlikely(pid < 0))
+   return 0;
+ } else if (vpid < 0) {
+  pid = vpid_to_pid(-vpid);
+  if (unlikely(pid < 0))
+   return 0;
+  pid = -pid;
+ }
+ return pid;
+}
+
+static inline int comb_pid_to_vpid(int pid)
+{
+ int vpid = pid;
+
+ if (pid > 0) {
+  vpid = pid_type_to_vpid(PIDTYPE_PID, pid);
+  if (unlikely(vpid < 0))
+   return 0;
+ } else if (pid < 0) {
+  vpid = pid_type_to_vpid(PIDTYPE_PGID, -pid);
```

```
+  if (unlikely(vpid < 0))
+   return 0;
+  vpid = -vpid;
+ }
+ return vpid;
+}
+
+extern int alloc_vpid(int pid, int vpid);
+extern void free_vpid(int vpid);
 #endif

 #define pid_task(elem, type) \
--- ./include/linux/sched.h.vpid_virt 2006-02-02 14:32:41.164807168 +0300
+++ ./include/linux/sched.h 2006-02-02 14:58:53.129832160 +0300
@@ -1327,6 +1327,80 @@ static inline pid_t get_task_ppid(struct
   return 0;
  return (p->pid > 1 ? p->group_leader->real_parent->tgid : 0);
 }
+#else
+static inline pid_t virt_pid(struct task_struct *tsk)
+{
+ return tsk->pids[PIDTYPE_PID].vnr;
+}
+
+static inline pid_t virt_tgid(struct task_struct *tsk)
+{
+ return tsk->pids[PIDTYPE_TGID].vnr;
+}
+
+static inline pid_t virt_pgid(struct task_struct *tsk)
+{
+ return tsk->pids[PIDTYPE_PGID].vnr;
+}
+
+static inline pid_t virt_sid(struct task_struct *tsk)
+{
+ return tsk->pids[PIDTYPE_SID].vnr;
+}
+
+static inline pid_t get_task_pid_ve(struct task_struct *tsk,
+  struct task_struct *ve_tsk)
+{
+ return task_inside_vps(ve_tsk) ? virt_pid(tsk) : tsk->pid;
+}
+
+static inline pid_t get_task_pid(struct task_struct *tsk)
+{
+ return inside_vps() ? virt_pid(tsk) : tsk->pid;
```

```
+}
+
+static inline pid_t get_task_tgid(struct task_struct *tsk)
+{
+ return inside_vps() ? virt_tgid(tsk) : tsk->pid;
+}
+
+static inline pid_t get_task_pgid(struct task_struct *tsk)
+{
+ return inside_vps() ? virt_pgid(tsk) : tsk->signal->pgrp;
+}
+
+static inline pid_t get_task_sid(struct task_struct *tsk)
+{
+ return inside_vps() ? virt_sid(tsk) : tsk->signal->session;
+}
+
+static inline int set_virt_pid(struct task_struct *tsk, pid_t pid)
+{
+ tsk->pids[PIDTYPE_PID].vnr = pid;
+ return pid;
+}
+
+static inline void set_virt_tgid(struct task_struct *tsk, pid_t pid)
+{
+ tsk->pids[PIDTYPE_TGID].vnr = pid;
+}
+
+static inline void set_virt_pgid(struct task_struct *tsk, pid_t pid)
+{
+ tsk->pids[PIDTYPE_PGID].vnr = pid;
+}
+
+static inline void set_virt_sid(struct task_struct *tsk, pid_t pid)
+{
+ tsk->pids[PIDTYPE_SID].vnr = pid;
+}
+
+static inline pid_t get_task_ppid(struct task_struct *p)
+{
+ if (!pid_alive(p))
+  return 0;
+ return (get_task_pid(p) > 1) ? get_task_pid(p->real_parent) : 0;
+}
 #endif

 /* set thread flags in other task's structures
--- ./kernel/pid.c.vpid_virt 2006-02-02 14:15:35.165782728 +0300
```

```
+++ ./kernel/pid.c 2006-02-02 14:58:34.632644160 +0300
@@ -27,6 +27,14 @@
 #include <linux/bootmem.h>
 #include <linux/hash.h>

+#ifdef CONFIG_VIRTUAL_PIDS
+static void __free_vpid(int vpid, struct task_struct *ve_tsk);
+#define PIDMAP_NRFREE (BITS_PER_PAGE / 2)
+#else
+#define __free_vpid(vpid, tsk) do { } while (0)
+#define PIDMAP_NRFREE BITS_PER_PAGE
+#endif
+
 #define pid_hashfn(nr) hash_long((unsigned long)nr, pidhash_shift)
 static struct hlist_head *pid_hash[PIDTYPE_MAX];
 static int pidhash_shift;
@@ -58,7 +66,7 @@ typedef struct pidmap {
 } pidmap_t;

 static pidmap_t pidmap_array[PIDMAP_ENTRIES] =
-  { [ 0 ... PIDMAP_ENTRIES-1 ] = { ATOMIC_INIT(BITS_PER_PAGE), NULL } };
+  { [ 0 ... PIDMAP_ENTRIES-1 ] = { ATOMIC_INIT(PIDMAP_NRFREE), NULL } };

 static  __cacheline_aligned_in_smp DEFINE_SPINLOCK(pidmap_lock);

@@ -67,6 +75,7 @@ fastcall void free_pidmap(int pid)
  pidmap_t *map = pidmap_array + pid / BITS_PER_PAGE;
  int offset = pid & BITS_PER_PAGE_MASK;

+ BUG_ON(__is_virtual_pid(pid) || pid == 1);
  clear_bit(offset, map->page);
  atomic_inc(&map->nr_free);
 }
@@ -77,6 +86,8 @@ int alloc_pidmap(void)
  pidmap_t *map;

  pid = last + 1;
+ if (__is_virtual_pid(pid))
+  pid += VPID_DIV;
  if (pid >= pid_max)
   pid = RESERVED_PIDS;
  offset = pid & BITS_PER_PAGE_MASK;
@@ -107,6 +118,8 @@ int alloc_pidmap(void)
     }
     offset = find_next_offset(map, offset);
     pid = mk_pid(map, offset);
+    if (__is_virtual_pid(pid))
+     pid += VPID_DIV;
```

```
    /*
     * find_next_offset() found a bit, the pid from it
     * is in-bounds, and if we fell back to the last
@@ -127,6 +140,8 @@ int alloc_pidmap(void)
     break;
   }
   pid = mk_pid(map, offset);
+  if (__is_virtual_pid(pid))
+   pid += VPID_DIV;
  }
  return -1;
 }
@@ -201,6 +216,7 @@ void fastcall detach_pid(task_t *task, e
   if (tmp != type && find_pid(tmp, nr))
    return;

+ __free_vpid(task->pids[type].vnr, task);
  free_pidmap(nr);
 }

@@ -234,6 +250,9 @@ void switch_exec_pids(task_t *leader, ta

  leader->pid = leader->tgid = thread->pid;
  thread->pid = thread->tgid;
+ set_virt_tgid(leader, virt_pid(thread));
+ set_virt_pid(leader, virt_pid(thread));
+ set_virt_pid(thread, virt_tgid(thread));

  attach_pid(thread, PIDTYPE_PID, thread->pid);
  attach_pid(thread, PIDTYPE_TGID, thread->tgid);
@@ -247,6 +266,344 @@ void switch_exec_pids(task_t *leader, ta
  attach_pid(leader, PIDTYPE_SID, leader->signal->session);
 }

+#ifdef CONFIG_VIRTUAL_PIDS
+/* Virtual PID bits.
+ *
+ * At the moment all internal structures in kernel store real global pid.
+ * The only place, where virtual PID is used, is at user frontend. We
+ * remap virtual pids obtained from user to global ones (vpid_to_pid) and
+ * map globals to virtuals before showing them to user (virt_pid_type).
+ *
+ * We hold virtual PIDs inside struct pid, so map global -> virtual is easy.
+ */
+
+pid_t __pid_type_to_vpid(int type, pid_t pid)
+{
+ struct pid * p;
```

```
+
+ if (unlikely(is_virtual_pid(pid)))
+  return -1;
+
+ read_lock(&tasklist_lock);
+ p = find_pid(type, pid);
+ if (p) {
+  pid = p->vnr;
+ } else {
+  pid = -1;
+ }
+ read_unlock(&tasklist_lock);
+ return pid;
+}
+
+pid_t pid_type_to_vpid(int type, pid_t pid)
+{
+ int vpid;
+
+ if (unlikely(pid <= 0))
+  return pid;
+
+ BUG_ON(is_virtual_pid(pid));
+
+ if (!inside_vps())
+  return pid;
+
+ vpid = __pid_type_to_vpid(type, pid);
+ if (unlikely(vpid == -1)) {
+  /* It is allowed: global pid can be used everywhere.
+   * This can happen, when kernel remembers stray pids:
+   * signal queues, locks etc.
+   */
+  vpid = pid;
+ }
+ return vpid;
+}
+
+/* To map virtual pids to global we maintain special hash table.
+ *
+ * Mapping entries are allocated when a process with non-trivial
+ * mapping is forked, which is possible only after VE migrated.
+ * Mappings are destroyed, when a global pid is removed from global
+ * pidmap, which means we do not need to refcount mappings.
+ */
+
+static struct hlist_head *vpid_hash;
+
```

```
+struct vpid_mapping
+{
+ int     pid;
+ int     vpid;
+ int     vpsid;
+ struct hlist_node link;
+};
+
+static kmem_cache_t *vpid_mapping_cachep;
+
+static inline int vpid_hashfn(int vnr, int vpsid)
+{
+ return hash_long((unsigned long)(vnr + (vpsid << 16)), pidhash_shift);
+}
+
+struct vpid_mapping *__lookup_vpid_mapping(int vnr, int vpsid)
+{
+ struct hlist_node *elem;
+ struct vpid_mapping *map;
+
+ hlist_for_each_entry(map, elem,
+    &vpid_hash[vpid_hashfn(vnr, vpsid)], link) {
+  if (map->vpid == vnr && map->vpsid == vpsid)
+   return map;
+ }
+ return NULL;
+}
+
+/* __vpid_to_pid() is raw version of vpid_to_pid(). It is to be used
+ * only under tasklist_lock. In some places we must use only this version
+ * (f.e. __kill_pg_info is called under write lock!)
+ *
+ * Caller should pass virtual pid. This function returns an error, when
+ * seeing a global pid.
+ */
+int __vpid_to_pid(int pid)
+{
+ struct vpid_mapping *map;
+ vps_info_t vps;
+
+ if (unlikely(!is_virtual_pid(pid) || !inside_vps()))
+  return -1;
+
+ vps = current_vps();
+ if (!sparse_vpid(vps)) {
+  if (pid != 1)
+   return pid - VPID_DIV;
+  return vps->init_task->pid;
```

```
+ }
+
+ map = __lookup_vpid_mapping(pid, vps->id);
+ if (map)
+  return map->pid;
+ return -1;
+}
+
+int vpid_to_pid(int pid)
+{
+ /* User gave bad pid. It is his problem. */
+ if (unlikely(pid <= 0))
+  return pid;
+
+ if (!is_virtual_pid(pid))
+  return pid;
+
+ read_lock(&tasklist_lock);
+ pid = __vpid_to_pid(pid);
+ read_unlock(&tasklist_lock);
+ return pid;
+}
+
+/*
+ * In simple case we have trivial vpid-to-pid conversion rule:
+ * vpid == 1 -> vps->init_task->pid
+ * else         pid & ~VPID_DIV
+ *
+ * when things get more complex we need to allocate mappings...
+ */
+
+static int add_mapping(int pid, int vpid, int vpsid, struct hlist_head *cache)
+{
+ if (pid > 0 && vpid > 0 && !__lookup_vpid_mapping(vpid, vpsid)) {
+  struct vpid_mapping *m;
+
+  if (hlist_empty(cache)) {
+   m = kmem_cache_alloc(vpid_mapping_cachep, GFP_ATOMIC);
+   if (unlikely(m == NULL))
+    return -ENOMEM;
+  } else {
+   m = hlist_entry(cache->first, struct vpid_mapping,
+     link);
+   hlist_del(&m->link);
+  }
+  m->pid = pid;
+  m->vpid = vpid;
+  m->vpsid = vpsid;
```

```
+  hlist_add_head(&m->link,
+    &vpid_hash[vpid_hashfn(vpid, vpsid)]);
+ }
+ return 0;
+}
+
+static int switch_to_sparse_mapping(int pid, vps_info_t vps)
+{
+ struct hlist_head cache;
+ task_t *g, *t;
+ int pcount;
+ int err;
+
+ /* Transition happens under write_lock_irq, so we try to make
+  * it more reliable and fast preallocating mapping entries.
+  * pcounter may be not enough, we could have lots of orphaned
+  * process groups and sessions, which also require mappings.
+  */
+ INIT_HLIST_HEAD(&cache);
+ pcount = get_vps_tasks_num(vps);
+ err = -ENOMEM;
+ while (pcount > 0) {
+  struct vpid_mapping *m;
+  m = kmem_cache_alloc(vpid_mapping_cachep, GFP_KERNEL);
+  if (!m)
+   goto out;
+  hlist_add_head(&m->link, &cache);
+  pcount--;
+ }
+
+ write_lock_irq(&tasklist_lock);
+ err = 0;
+ if (sparse_vpid(vps))
+  goto out_sparse;
+
+ err = -ENOMEM;
+ do_each_thread(g, t) {
+  if (t->pid == pid)
+   continue;
+  if (add_mapping(t->pid, virt_pid(t), vps->id, &cache))
+   goto out_unlock;
+ } while_each_thread(g, t);
+
+ for_each_process(t) {
+  if (t->pid == pid)
+   continue;
+
+  if (add_mapping(t->tgid, virt_tgid(t), vps->id,
```

```
+     &cache))
+   goto out_unlock;
+  if (add_mapping(t->signal->pgrp, virt_pgid(t), vps->id,
+     &cache))
+   goto out_unlock;
+  if (add_mapping(t->signal->session, virt_sid(t), vps->id,
+     &cache))
+   goto out_unlock;
+ }
+ set_sparse_vpid(vps);
+ err = 0;
+
+out_unlock:
+ if (err) {
+  int i;
+
+  for (i=0; i<(1<<pidhash_shift); i++) {
+   struct hlist_node *elem, *next;
+   struct vpid_mapping *map;
+
+   hlist_for_each_entry_safe(map, elem, next, &vpid_hash[i], link) {
+    if (map->vpsid == vps->id) {
+     hlist_del(elem);
+     hlist_add_head(elem, &cache);
+    }
+   }
+  }
+ }
+out_sparse:
+ write_unlock_irq(&tasklist_lock);
+
+out:
+ while (!hlist_empty(&cache)) {
+  struct vpid_mapping *m;
+  m = hlist_entry(cache.first, struct vpid_mapping, link);
+  hlist_del(&m->link);
+  kmem_cache_free(vpid_mapping_cachep, m);
+ }
+ return err;
+}
+
+int alloc_vpid(int pid, int virt_pid)
+{
+ int result;
+ struct vpid_mapping *m;
+ vps_info_t vps;
+
+ if (!inside_vps())
```

```
+  return pid;
+
+ vps = current_vps();
+ if (!sparse_vpid(vps)) {
+  if (virt_pid == -1)
+   return pid + VPID_DIV;
+
+  if (virt_pid == 1 || virt_pid == pid + VPID_DIV)
+   return virt_pid;
+
+  if ((result = switch_to_sparse_mapping(pid, vps)) < 0)
+   return result;
+ }
+
+ m = kmem_cache_alloc(vpid_mapping_cachep, GFP_KERNEL);
+ if (!m)
+  return -ENOMEM;
+
+ m->pid = pid;
+ m->vpsid = vps->id;
+
+ result = (virt_pid == -1) ? pid + VPID_DIV : virt_pid;
+
+ write_lock_irq(&tasklist_lock);
+ if (unlikely(__lookup_vpid_mapping(result, m->vpsid))) {
+  if (virt_pid > 0) {
+   result = -EEXIST;
+   goto out;
+  }
+
+  /* No luck. Now we search for some not-existing vpid.
+   * It is weak place. We do linear search. */
+  do {
+   result++;
+   if (!__is_virtual_pid(result))
+    result += VPID_DIV;
+   if (result >= pid_max)
+    result = RESERVED_PIDS + VPID_DIV;
+  } while (__lookup_vpid_mapping(result, m->vpsid) != NULL);
+
+  /* And set last_pid in hope future alloc_pidmap to avoid
+   * collisions after future alloc_pidmap() */
+  last_pid = result - VPID_DIV;
+ }
+ if (result > 0) {
+  m->vpid = result;
+  hlist_add_head(&m->link,
+    &vpid_hash[vpid_hashfn(result, m->vpsid)]);
```

```
+ }
+out:
+ write_unlock_irq(&tasklist_lock);
+ if (result < 0)
+  kmem_cache_free(vpid_mapping_cachep, m);
+ return result;
+}
+
+static void __free_vpid(int vpid, struct task_struct *ve_tsk)
+{
+ struct vpid_mapping *m;
+ vps_info_t vps;
+
+ if (!__is_virtual_pid(vpid) && (vpid != 1 || !task_inside_vps(ve_tsk)))
+  return;
+
+ vps = task_vps(ve_tsk);
+ if (!sparse_vpid(vps))
+  return;
+
+ m = __lookup_vpid_mapping(vpid, vps->id);
+ BUG_ON(m == NULL);
+ hlist_del(&m->link);
+ kmem_cache_free(vpid_mapping_cachep, m);
+}
+EXPORT_SYMBOL(alloc_vpid);
+
+void free_vpid(int vpid)
+{
+ write_lock_irq(&tasklist_lock);
+ __free_vpid(vpid, current);
+ write_unlock_irq(&tasklist_lock);
+}
+
+#endif
+
 /*
  * The pid hash table is scaled according to the amount of memory in the
  * machine.  From a minimum of 16 slots up to 4096 slots at one gigabyte or
@@ -273,6 +630,14 @@ void __init pidhash_init(void)
   for (j = 0; j < pidhash_size; j++)
    INIT_HLIST_HEAD(&pid_hash[i][j]);
  }
+
+#ifdef CONFIG_VIRTUAL_PIDS
+ vpid_hash = alloc_bootmem(pidhash_size * sizeof(struct hlist_head));
+ if (!vpid_hash)
+  panic("Could not alloc vpid_hash!\n");
```

```
+ for (j = 0; j < pidhash_size; j++)
+  INIT_HLIST_HEAD(&vpid_hash[j]);
+#endif
 }

 void __init pidmap_init(void)
@@ -289,4 +654,12 @@ void __init pidmap_init(void)

  for (i = 0; i < PIDTYPE_MAX; i++)
   attach_pid(current, i, 0);
+
+#ifdef CONFIG_VIRTUAL_PIDS
+ vpid_mapping_cachep =
+  kmem_cache_create("vpid_mapping",
+    sizeof(struct vpid_mapping),
+    __alignof__(struct vpid_mapping),
+    SLAB_PANIC, NULL, NULL);
+#endif
 }
```