Subject: [PATCH v4] posix timers: allocate timer id per process
Posted by Stanislav Kinsbursky on Fri, 19 Oct 2012 07:49:39 GMT
View Forum Message <> Reply to Message

v4:
1) a couple of coding style fixes (lines over 80 characters)

v3:
1) hash calculation simlified to improve perfomance.

v2:
1) Hash table become RCU-friendly. Hash table search now done under RCU lock protection.
I've tested scalability on KVM with 4 CPU. The testing environment was build of 10 processes, each had 512 posix timers running (SIGSEV_NONE) and was calling timer_gettime() in loop. With all this stuff being running, I was measuring time of calling of syscall timer_gettime() 10000 times.

Without this patch: ~7ms
With this patch   : ~7ms

This patch is required CRIU project (www.criu.org).
To migrate processes with posix timers we have to make sure, that we can restore posix timer with proper id.
Currently, this is not true, because timer ids are allocated globally.
So, this is precursor patch and it's purpose is make posix timer id to be allocated per process.

Patch replaces global idr with global hash table for posix timers and makes timer ids unique not globally, but per task. Next free timer id is type of integer and stored on signal struct (posix_timer_id). If free timer id reaches negative value on timer creation, it will be dropped to zero and -EAGAIN will be returned to user.
Hash table is size of page (4KB).
Key is constructed as follows:
key = hash_32(current->signal) ^ hash_32(posix_timer_id);

Signed-off-by: Stanislav Kinsbursky <skinsbursky@parallels.com>
---
 include/linux/posix-timers.h | 1
 include/linux/sched.h | 4 +
 kernel/posix-timers.c | 113 +++++++++++++++++++++++++++++---------------
 3 files changed, 79 insertions(+), 39 deletions(-)

diff --git a/include/linux/posix-timers.h b/include/linux/posix-timers.h
index 042058f..60bac69 100644
--- a/include/linux/posix-timers.h
+++ b/include/linux/posix-timers.h

```
@@ -55,6 +55,7 @@ struct cpu_timer_list {
 /* POSIX.1b interval timer structure. */
 struct k_itimer {
  struct list_head list;  /* free/ allocate list */
+ struct hlist_node t_hash;
  spinlock_t it_lock;
  clockid_t it_clock;  /* which timer type */
  timer_t it_id;   /* timer id */
diff --git a/include/linux/sched.h b/include/linux/sched.h
index 0dd42a0..dce1651 100644
--- a/include/linux/sched.h
+++ b/include/linux/sched.h
@@ -51,6 +51,7 @@ struct sched_param {
 #include <linux/cred.h>
 #include <linux/llist.h>
 #include <linux/uidgid.h>
+#include <linux/idr.h>

 #include <asm/processor.h>

@@ -536,7 +537,8 @@ struct signal_struct {
  unsigned int  has_child_subreaper:1;

  /* POSIX.1b Interval Timers */
- struct list_head posix_timers;
+ int   posix_timer_id;
+ struct list_head posix_timers;

  /* ITIMER_REAL timer for the process */
  struct hrtimer real_timer;
diff --git a/kernel/posix-timers.c b/kernel/posix-timers.c
index 69185ae..6d94d8e 100644
--- a/kernel/posix-timers.c
+++ b/kernel/posix-timers.c
@@ -47,31 +47,28 @@
 #include <linux/wait.h>
 #include <linux/workqueue.h>
 #include <linux/export.h>
+#include <linux/hash.h>

 /*
- * Management arrays for POSIX timers.  Timers are kept in slab memory
- * Timer ids are allocated by an external routine that keeps track of the
- * id and the timer.  The external interface is:
- *
- * void *idr_find(struct idr *idp, int id);           to find timer_id <id>
- * int idr_get_new(struct idr *idp, void *ptr);       to get a new id and
- *                                    related it to <ptr>
```

```
- * void idr_remove(struct idr *idp, int id);          to release <id>
- * void idr_init(struct idr *idp);                    to initialize <idp>
- *                                          which we supply.
- * The idr_get_new *may* call slab for more memory so it must not be
- * called under a spin lock.  Likewise idr_remore may release memory
- * (but it may be ok to do this under a lock...).
- * idr_find is just a memory look up and is quite fast.  A -1 return
- * indicates that the requested id does not exist.
+ * Management arrays for POSIX timers. Timers are now kept in static PAGE-size
+ * hash table.
+ * Timer ids are allocated by local routine, which selects proper hash head by
+ * key, constructed from current->signal address and per signal struct counter.
+ * This keeps timer ids unique per process, but now they can intersect between
+ * processes.
  */

 /*
  * Lets keep our timers in a slab cache :-)
  */
 static struct kmem_cache *posix_timers_cache;
-static struct idr posix_timers_id;
-static DEFINE_SPINLOCK(idr_lock);
+
+#define POSIX_TIMERS_HASH_BITS  9
+#define POSIX_TIMERS_HASH_SIZE  (1 << POSIX_TIMERS_HASH_BITS)
+
+/* Hash table is size of PAGE currently */
+static struct hlist_head posix_timers_hashtable[POSIX_TIMERS_HASH_SIZE];
+static DEFINE_SPINLOCK(hash_lock);

 /*
  * we assume that the new SIGEV_THREAD_ID shares no bits with the other
@@ -152,6 +149,57 @@ static struct k_itimer *__lock_timer(timer_t timer_id, unsigned long
*flags);
  __timr;          \
 })

+static int hash(struct signal_struct *sig, unsigned int nr)
+{
+ return hash_32(hash32_ptr(sig) ^ nr, POSIX_TIMERS_HASH_BITS);
+}
+
+static struct k_itimer *__posix_timers_find(struct hlist_head *head,
+         struct signal_struct *sig,
+         timer_t id)
+{
+ struct hlist_node *node;
+ struct k_itimer *timer;
```

```
+
+ hlist_for_each_entry_rcu(timer, node, head, t_hash) {
+  if ((timer->it_signal == sig) && (timer->it_id == id))
+   return timer;
+ }
+ return NULL;
+}
+
+static struct k_itimer *posix_timer_by_id(timer_t id)
+{
+ struct signal_struct *sig = current->signal;
+ struct hlist_head *head = &posix_timers_hashtable[hash(sig, id)];
+
+ return __posix_timers_find(head, sig, id);
+}
+
+static int posix_timer_add(struct k_itimer *timer)
+{
+ struct signal_struct *sig = current->signal;
+ int next_free_id = sig->posix_timer_id;
+ struct hlist_head *head;
+ int ret = -ENOENT;
+
+ do {
+  spin_lock(&hash_lock);
+  head = &posix_timers_hashtable[hash(sig, sig->posix_timer_id)];
+  if (!__posix_timers_find(head, sig, sig->posix_timer_id)) {
+   hlist_add_head_rcu(&timer->t_hash, head);
+   ret = sig->posix_timer_id++;
+  } else {
+   if (++sig->posix_timer_id < 0)
+    sig->posix_timer_id = 0;
+   if (sig->posix_timer_id == next_free_id)
+    ret = -EAGAIN;
+  }
+  spin_unlock(&hash_lock);
+ } while (ret == -ENOENT);
+ return ret;
+}
+
 static inline void unlock_timer(struct k_itimer *timr, unsigned long flags)
 {
  spin_unlock_irqrestore(&timr->it_lock, flags);
@@ -271,6 +319,7 @@ static __init int init_posix_timers(void)
   .timer_get = common_timer_get,
   .timer_del = common_timer_del,
  };
+ int i;
```

```
  posix_timers_register_clock(CLOCK_REALTIME, &clock_realtime);
  posix_timers_register_clock(CLOCK_MONOTONIC, &clock_monotonic);
@@ -282,7 +331,8 @@ static __init int init_posix_timers(void)
  posix_timers_cache = kmem_cache_create("posix_timers_cache",
      sizeof (struct k_itimer), 0, SLAB_PANIC,
      NULL);
- idr_init(&posix_timers_id);
+ for (i = 0; i < POSIX_TIMERS_HASH_SIZE; i++)
+  INIT_HLIST_HEAD(&posix_timers_hashtable[i]);
  return 0;
 }

@@ -504,9 +554,9 @@ static void release_posix_timer(struct k_itimer *tmr, int it_id_set)
 {
  if (it_id_set) {
   unsigned long flags;
-  spin_lock_irqsave(&idr_lock, flags);
-  idr_remove(&posix_timers_id, tmr->it_id);
-  spin_unlock_irqrestore(&idr_lock, flags);
+  spin_lock_irqsave(&hash_lock, flags);
+  hlist_del_rcu(&tmr->t_hash);
+  spin_unlock_irqrestore(&hash_lock, flags);
  }
  put_pid(tmr->it_pid);
  sigqueue_free(tmr->sigq);
@@ -552,22 +602,9 @@ SYSCALL_DEFINE3(timer_create, const clockid_t, which_clock,
   return -EAGAIN;

  spin_lock_init(&new_timer->it_lock);
- retry:
- if (unlikely(!idr_pre_get(&posix_timers_id, GFP_KERNEL))) {
-  error = -EAGAIN;
-  goto out;
- }
- spin_lock_irq(&idr_lock);
- error = idr_get_new(&posix_timers_id, new_timer, &new_timer_id);
- spin_unlock_irq(&idr_lock);
- if (error) {
-  if (error == -EAGAIN)
-   goto retry;
-  /*
-   * Weird looking, but we return EAGAIN if the IDR is
-   * full (proper POSIX return value for this)
-   */
-  error = -EAGAIN;
+ new_timer_id = posix_timer_add(new_timer);
+ if (new_timer_id < 0) {
```

```
+  error = new_timer_id;
   goto out;
  }

@@ -640,7 +677,7 @@ static struct k_itimer *__lock_timer(timer_t timer_id, unsigned long *flags)
  struct k_itimer *timr;

  rcu_read_lock();
- timr = idr_find(&posix_timers_id, (int)timer_id);
+ timr = posix_timer_by_id(timer_id);
  if (timr) {
   spin_lock_irqsave(&timr->it_lock, *flags);
   if (timr->it_signal == current->signal) {
```

Subject: Re: [PATCH v4] posix timers: allocate timer id per process
Posted by Eric Dumazet on Fri, 19 Oct 2012 07:56:40 GMT
View Forum Message <> Reply to Message

On Fri, 2012-10-19 at 11:50 +0400, Stanislav Kinsbursky wrote:
> v4:
> 1) a couple of coding style fixes (lines over 80 characters)
>
> v3:
> 1) hash calculation simlified to improve perfomance.
>
> v2:
> 1) Hash table become RCU-friendly. Hash table search now done under RCU lock
> protection.

This should not be in the changelog, only after the --- separator.

> I've tested scalability on KVM with 4 CPU. The testing environment was build
> of 10 processes, each had 512 posix timers running (SIGSEV_NONE) and was
> calling timer_gettime() in loop. With all this stuff being running, I was
> measuring time of calling of syscall timer_gettime() 10000 times.
>
> Without this patch: ~7ms
> With this patch   : ~7ms
>
> This patch is required CRIU project (www.criu.org).
> To migrate processes with posix timers we have to make sure, that we can
> restore posix timer with proper id.
> Currently, this is not true, because timer ids are allocated globally.
> So, this is precursor patch and it's purpose is make posix timer id to be
> allocated per process.
>
> Patch replaces global idr with global hash table for posix timers and

> makes timer ids unique not globally, but per task. Next free timer id is type
> of integer and stored on signal struct (posix_timer_id). If free timer id
> reaches negative value on timer creation, it will be dropped to zero and
> -EAGAIN will be returned to user.

I wonder if some applications relied on our idr, assuming they would get
low values for their timer id.
(We could imagine some applications use a table indexed by the timer id)

> Hash table is size of page (4KB).

Only on x86_64. Why not instead saying hashtable has 512 slots ?

> Key is constructed as follows:
> key = hash_32(current->signal) ^ hash_32(posix_timer_id);

This is outdated.

>
> Signed-off-by: Stanislav Kinsbursky <skinsbursky@parallels.com>
> ---

Thanks

Subject: Re: [PATCH v4] posix timers: allocate timer id per process
Posted by Stanislav Kinsbursky on Fri, 19 Oct 2012 09:38:46 GMT
View Forum Message <> Reply to Message

> I wonder if some applications relied on our idr, assuming they would get
> low values for their timer id.
> (We could imagine some applications use a table indexed by the timer id)

Hmm.
Probably, this particular case can be optimised by tuning min_id to id of
releasing timer (if id of this timer is less than current->signal min_id).
Does this approach solves the issue you mentioned above?

--
Best regards,
Stanislav Kinsbursky

Subject: Re: [PATCH v4] posix timers: allocate timer id per process
Posted by Eric Dumazet on Fri, 19 Oct 2012 10:43:03 GMT
View Forum Message <> Reply to Message

On Fri, 2012-10-19 at 13:38 +0400, Stanislav Kinsbursky wrote:

> > I wonder if some applications relied on our idr, assuming they would get
> > low values for their timer id.
> > (We could imagine some applications use a table indexed by the timer id)
>
> Hmm.
> Probably, this particular case can be optimised by tuning min_id to id of
> releasing timer (if id of this timer is less than current->signal min_id).
> Does this approach solves the issue you mentioned above?

Not generally, but I am not sure we want a per signal_struct idr ;)

Really that should be clearly explained in the changelog, so that buggy applications can have a clue of what happened.

When we changed UDP source port selection being random instead of sequential, maybe this broke some applications. That was an implementation choice (with security impact).
