Subject: [PATCH 11/15] Signal semantics
Posted by Pavel Emelianov on Thu, 26 Jul 2007 14:55:03 GMT
View Forum Message <> Reply to Message

From: Sukadev Bhattiprolu <sukadev@us.ibm.com>

With support for multiple pid namespaces, each pid namespace has a
separate child reaper and this process needs some special handling
of signals.

- The child reaper should appear like a normal process to other
  processes in its ancestor namespaces and so should be killable
  (or not) in the usual way.

    - The child reaper should receive, from processes in it's active
      and decendent namespaces, only those signals for which it has
      installed a signal handler.

- System-wide signals (eg: kill signum -1) from within a child namespace
  should only affect processes within that namespace and descendant
  namespaces. They should not be posted to processes in ancestor or
  sibling namespaces.

- If the sender of a signal does not have a pid_t in the receiver's
  namespace (eg: a process in init_pid_ns sends a signal to a process
  in a descendant namespace), the sender's pid should appear as 0
  in the signal's siginfo structure.

- Existing rules for SIGIO delivery still apply and a process can
  choose any other process in its namespace and descendant namespaces
  to receive the SIGIO signal.

  The following appears to be incorrect in the fcntl() man page for
  F_SETOWN.

> Sending a signal to the owner process (group) specified by
> F_SETOWN is subject to the same permissions checks as are
> described for kill(2), where the sending process is the one that
> employs F_SETOWN (but see BUGS below).

Current behavior is that the SIGIO signal is delivered on behalf of
the process that caused the event (eg: made data available on the
file) and not the process that called fcntl().

To implement the above requirements, we:

- Add a check in check_kill_permission() for a process within a

namespace sending the fast-pathed, SIGKILL signal.

- We use a flag, SIGQUEUE_CINIT, to tell the container-init if
a signal posted to its queue is from a process within its own
namespace. The flag is set in send_signal() if a process attempts
to send a signal to its container-init.

The SIGQUEUE_CINIT flag is checked in collect_signal() - if
the flag is set, collect_signal() sets the KERN_SIGINFO_CINIT
flag in the kern_siginfo. The KERN_SIGINFO_CINIT flag indicates
that the sender is from within the namespace and the container-init
can choose to ignore the signal.

If the KERN_SIGINFO_CINIT flag is clear in get_signal_to_deliver(),
the signal originated from an ancestor namespace and so the
container-init honors the signal.

Note: We currently use two flags, SIGQUEUE_CINIT, KERN_SIGINFO_CINIT to
avoid modifying 'struct sigqueue'. If 'kern_siginfo' approach is
feasible, we could use 'kern_siginfo' in sigqueue and eliminate
SIGQUEUE_CINIT.

Signed-off-by: Sukadev Bhattiprolu <sukadev@us.ibm.com>
Signed-off-by: Pavel Emelyanov <xemul@openvz.org>

---

include/linux/pid.h | 3 ++
include/linux/signal.h | 1
kernel/pid.c | 46 ++
kernel/signal.c | 63 ++-
4 files changed, 112 insertions(+), 1 deletion(-)

diff -upr linux-2.6.23-rc1-mm1.orig/include/linux/pid.h linux-2.6.23-rc1-mm1-7/include/linux/pid.h
--- linux-2.6.23-rc1-mm1.orig/include/linux/pid.h 2007-07-26 16:34:45.000000000 +0400
+++ linux-2.6.23-rc1-mm1-7/include/linux/pid.h 2007-07-26 16:36:37.000000000 +0400
@@ -71,6 +77,9 @@ extern struct task_struct *FASTCALL(pid_
extern struct task_struct *FASTCALL(get_pid_task(struct pid *pid,
enum pid_type));

+extern int task_visible_in_pid_ns(struct task_struct *tsk,
+ struct pid_namespace *ns);
+extern int pid_ns_equal(struct task_struct *tsk);
extern struct pid *get_task_pid(struct task_struct *task, enum pid_type type);

/*
diff -upr linux-2.6.23-rc1-mm1.orig/include/linux/signal.h

```
linux-2.6.23-rc1-mm1-7/include/linux/signal.h
--- linux-2.6.23-rc1-mm1.orig/include/linux/signal.h 2007-07-26 16:34:45.000000000 +0400
+++ linux-2.6.23-rc1-mm1-7/include/linux/signal.h 2007-07-26 16:36:37.000000000 +0400
@@ -20,6 +27,7 @@ struct sigqueue {

 /* flags values. */
 #define SIGQUEUE_PREALLOC 1
+#define SIGQUEUE_CINIT  2

 struct sigpending {
  struct list_head list;
diff -upr linux-2.6.23-rc1-mm1.orig/kernel/pid.c linux-2.6.23-rc1-mm1-7/kernel/pid.c
--- linux-2.6.23-rc1-mm1.orig/kernel/pid.c 2007-07-26 16:34:45.000000000 +0400
+++ linux-2.6.23-rc1-mm1-7/kernel/pid.c 2007-07-26 16:36:37.000000000 +0400
@@ -318,6 +355,52 @@ struct task_struct * fastcall pid_task(s
 }

 /*
+ * Return TRUE if the task @p is visible in the pid namespace @ns
+ *
+ * Note: @p is visible in @ns if the active-pid-ns of @p is either equal to
+ *  @ns or is a descendant of @ns.
+ *
+ *  @p is not visible in @ns if active-pid-ns of @p is an ancestor of @ns.
+ *  Eg: Processes in init-pid-ns are not visible in child pid namespaces.
+ *  They should not receive any system-wide signals from a child-pid-
+ *  namespace for instance.
+ */
+int task_visible_in_pid_ns(struct task_struct *p, struct pid_namespace *ns)
+{
+ int i;
+ struct pid *pid = task_pid(p);
+
+ if (!pid)
+  return 0;
+
+ for (i = 0; i <= pid->level; i++) {
+  if (pid->numbers[i].ns == ns)
+   return 1;
+ }
+
+ return 0;
+}
+EXPORT_SYMBOL(task_visible_in_pid_ns);
+
+/*
+ * Return TRUE if the active pid namespace of @tsk is same as active
+ * pid namespace of 'current'
```

```
+ */
+
+static inline struct pid_namespace *pid_active_ns(struct pid *pid)
+{
+ if (pid == NULL)
+  return NULL;
+
+ return pid->numbers[pid->level].ns;
+}
+
+int pid_ns_equal(struct task_struct *tsk)
+{
+ return pid_active_ns(task_pid(tsk)) == pid_active_ns(task_pid(current));
+}
+
+/*
  * Must be called under rcu_read_lock() or with tasklist_lock read-held.
  */
 struct task_struct *find_task_by_pid_type_ns(int type, int nr,
diff -upr linux-2.6.23-rc1-mm1.orig/kernel/signal.c linux-2.6.23-rc1-mm1-7/kernel/signal.c
--- linux-2.6.23-rc1-mm1.orig/kernel/signal.c 2007-07-26 16:34:45.000000000 +0400
+++ linux-2.6.23-rc1-mm1-7/kernel/signal.c 2007-07-26 16:36:37.000000000 +0400
@@ -323,6 +325,9 @@ static int collect_signal(int sig, struc
  if (first) {
   list_del_init(&first->list);
   copy_siginfo(info, &first->info);
+  if (first->flags & SIGQUEUE_CINIT)
+   kinfo->flags |= KERN_SIGINFO_CINIT;
+
   __sigqueue_free(first);
   if (!still_pending)
    sigdelset(&list->signal, sig);
@@ -343,6 +348,8 @@ static int collect_signal(int sig, struc
 {
  int sig = next_signal(pending, mask);

+ kinfo->flags &= ~KERN_SIGINFO_CINIT;
+
  if (sig) {
   if (current->notifier) {
    if (sigismember(current->notifier_mask, sig)) {
@@ -522,6 +547,20 @@ static int rm_from_queue(unsigned long m
  return 1;
 }

+static int deny_signal_to_container_init(struct task_struct *tsk, int sig)
+{
+ /*
```

```
+  * If receiver is the container-init of sender and signal is SIGKILL
+  * reject it right-away. If signal is any other one, let the container
+  * init decide (in get_signal_to_deliver()) whether to handle it or
+  * ignore it.
+  */
+ if (is_container_init(tsk) && (sig == SIGKILL) && pid_ns_equal(tsk))
+  return -EPERM;
+
+ return 0;
+}
+
 /*
  * Bad permissions for sending the signal
  */
@@ -545,6 +584,10 @@ static int check_kill_permission(int sig
      && !capable(CAP_KILL))
   return error;

+ error = deny_signal_to_container_init(t, sig);
+ if (error)
+  return error;
+
  return security_task_kill(t, info, sig, 0);
 }

@@ -659,6 +702,34 @@ static void handle_stop_signal(int sig,
  }
 }

+static void encode_sender_info(struct task_struct *t, struct sigqueue *q)
+{
+ /*
+  * If sender (i.e 'current') and receiver have the same active
+  * pid namespace and the receiver is the container-init, set the
+  * SIGQUEUE_CINIT flag. This tells the container-init that the
+  * signal originated in its own namespace and so it can choose
+  * to ignore the signal.
+  *
+  * If the receiver is the container-init of a pid namespace,
+  * but the sender is from an ancestor pid namespace, the
+  * container-init cannot ignore the signal. So clear the
+  * SIGQUEUE_CINIT flag in this case.
+  *
+  * Also, if the sender does not have a pid_t in the receiver's
+  * active pid namespace, set si_pid to 0 and pretend it originated
+  * from the kernel.
+  */
+ if (pid_ns_equal(t)) {
```

```
+  if (is_container_init(t)) {
+   q->flags |= SIGQUEUE_CINIT;
+  }
+ } else {
+  q->info.si_pid = 0;
+  q->info.si_code = SI_KERNEL;
+ }
+}
+
 static int send_signal(int sig, struct siginfo *info, struct task_struct *t,
    struct sigpending *signals)
 {
@@ -710,6 +781,7 @@ static int send_signal(int sig, struct s
    copy_siginfo(&q->info, info);
    break;
   }
+  encode_sender_info(t, q);
  } else if (!is_si_special(info)) {
   if (sig >= SIGRTMIN && info->si_code != SI_USER)
   /*
@@ -1149,6 +1221,8 @@ EXPORT_SYMBOL_GPL(kill_pid_info_as_uid);
 static int kill_something_info(int sig, struct siginfo *info, int pid)
 {
  int ret;
+ struct pid_namespace *my_ns = task_active_pid_ns(current);
+
  rcu_read_lock();
  if (!pid) {
   ret = kill_pgrp_info(sig, info, task_pgrp(current));
@@ -1158,6 +1232,13 @@ static int kill_something_info(int sig,

   read_lock(&tasklist_lock);
   for_each_process(p) {
+   /*
+    * System-wide signals apply only to the sender's
+    * pid namespace, unless issued from init_pid_ns.
+    */
+   if (!task_visible_in_pid_ns(p, my_ns))
+    continue;
+
    if (p->pid > 1 && p->tgid != current->tgid) {
     int err = group_send_sig_info(sig, info, p);
     ++count;
@@ -1852,7 +1950,7 @@ relock:
    * within that pid space. It can of course get signals from
    * its parent pid space.
    */
-  if (current == task_child_reaper(current))
```

```
+  if (kinfo.flags & KERN_SIGINFO_CINIT)
     continue;

   if (sig_kernel_stop(signr)) {
```