Subject: [PATCH 00/17] Pid-NS(V3) Enable multiple pid namespaces
Posted by Sukadev Bhattiprolu on Sat, 16 Jun 2007 22:57:46 GMT
View Forum Message <> Reply to Message

Date: Sat, 16 Jun 2007 15:29:14 -0700
From: sukadev@linux.vnet.ibm.com
Subject: [PATCH 00/17] Pid-NS(V3) Enable multiple pid namespaces

--
See Changelog below for changes since Pid-NS(V2) (2.6.21-mm2-pidns3).

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

A pid namespace is a "view" of a particular set of tasks on the system.
They work in a similar way to filesystem namespaces. A file (or a
process) can be accessed in multiple namespaces, but it may have a
different name in each. In a filesystem, this name might be /etc/passwd
in one namespace, but /chroot/etc/passwd in another.

For processes, a process may have pid 1234 in one namespace, but be pid
1 in another. This allows new pid namespaces to have basically
arbitrary pids, and not have to worry about what pids exist in other
namespaces. This is essential for checkpoint/restart where a restarted
process's pid might collide with an existing process on the system's
pid.

In this particular implementation, pid namespaces have a parent-child
relationship, just like processes. A process in a pid namespace may see
all of the processes in the same namespace, as well as all of the
processes in all of the namespaces which are children of its namespace.
Processes may not, however, see others which are in their parent's
namespace, but not in their own. The same goes for sibling namespaces.

Imagine it like a directory structure that has several processes
chroot'd into it:

```
/
|-- dir0
|   |-- subdir0
|   `-- subdir1
`-- dir1
    |-- subdir2
    `-- subdir3
```

A process in the '/' dir can see all of the directories. However, one
chrooted into dir0 can only see subdirs 0 and 1. It can not see dir1,

or subdirs 2 and 3. Pid namespaces work in the same way.

Following are the patches in this set. They apply to the 2.6.22-rc4-mm2 kernel.

[PATCH 1/17] Define and use task_active_pid_ns() wrapper

[PATCH 2/17] Rename pid_nr function

[PATCH 3/17] Rename child_reaper function.

[PATCH 4/17] Use pid_to_nr() in process info functions

[PATCH 5/17] Use task_pid() to find leader's pid

[PATCH 6/17] Define is_global_init() and is_container_init()

[PATCH 7/17] Move alloc_pid call to copy_process

[PATCH 8/17] Define and use pid->upid_list list.

[PATCH 9/17] Use pid ns from upid_list list

[PATCH 10/17] Define CLONE_NEWPID flag

[PATCH 11/17] Enable cloning pid namespace

[PATCH 12/17] Terminate processes in a ns when reaper is exiting.

[PATCH 13/17] Inline some pid functions and cleanup headers

[PATCH 14/17] Allow task_active_pid_ns() to be NULL

[PATCH 15/17] Signal support with pid namespaces

[PATCH 16/17] Remove proc_mnt's use for killing inodes

[PATCH 17/17] Introduce proc_mnt for pid_ns

Changelog:

(Summary of major changes since last post to Containers@).
These also listed in changelog of relevant patches.

2.6.22-rc4-mm2-pidns2:

- CLONE_NEWPID value has changed to 0x20000000 and so the pidns_exec.c
  example from previous version should be updated (or attached

pidns_exec.c must be used).

- Implement signal support with child pid namespaces.

- Modify is_global_init() implementation to use a global variable

- Bugfix: leaking 'struct pids'

- Limit to and optimize for MAX_NESTED_PID_NS levels of nested pid namespaces which is currently set to 4. Support for deeper nesting requires trivial changes to the macro and to a global variable, pid_cahce_names[], and a kernel rebuild. clone() fails with -EINVAL when attempting to exceed MAX_NESTED_PID_NS levels of nesting.

2.6.21-mm2-pidns3:

- Rename some functions for clarity: child_reaper() to task_child_reaper(), pid_nr() to pid_to_nr(), task_pid_ns() to task_active_pid_ns().

- Define/use helpers to access parent process info: task_parent_tgid(), task_parent_pid(), task_tracer_pid() task_tracer_tgid().

- 'struct upid' used to be called 'struct pid_nr' and a list of these were hanging off of 'struct pid'. So, renamed 'struct pid_nr' and now hold them in a statically sized array in 'struct pid' since the number of 'struct upid's for a process is known at process-creation time.

- Allow terminating the child reaper (and hence the entire namespace) from an ancestor namespace.

Dictionary of some process data structures/functions.

struct pid:
Kernel layer process id with lightweight reference to process.

struct pid_link:
Connects the task_struct of a process to its struct pids

pid_t:
User space layer integer process id

struct pid_namespace:
Process id namespace

struct upid:
Connects a pid_t with the pid_ns in which it is valid. Each

'struct pid' has a list of these (pid->upid_list) representing
every pid_t by which it is known in user space.

pid_to_nr():
Function call (used to be pid_nr()) - return the pid_t of a
process depending on the pid namespace of caller.

pidmap:
A bitmap representing which pid_t values are in use/free.
Used to efficiently assign a pid_t to newly created processes.

TODO:
As of 2.6.22-rc4-mm2-pidns1:

1. Include pid_namespace in pid_hash() so processes with same pid_t
   in different namespaces are on different hash lists.

2. Sending SIGKILL to the child reaper of a namespace terminates the
   namespace. But if the namespace remounted /proc from userspace,
   /proc would remain mounted even after reaper and other process in
   the namespace go away.

3. Identify clone flags that should not be specified with CLONE_NEWPID
   and return -EINVAL from copy_process(), if they are specified.

4. Add privilege check for CLONE_NEWPID.

5. Do we need a lock in task_child_reaper() for any race with do_exit()

6. Consider maintaining a per-pid namespace tasklist. Use that list
   to terminate processes in the namespace more efficiently. Such a
   tasklist may also be useful to freeze or checkpoint an application.

To test/create pidnamespaces, save the C program attached below as
pidns_exec.c and run following commands:

$ make pidns_exec

Test1: Create a shell in a new pid namespace and ensure only processes
in that namespace are visible from the namespace.

$ cat lxc-wrap.sh
#!/bin/sh

if [$$ -eq 1]; then
 echo "$$ (`basename $0`): Executing in nested pid ns..."
fi

```
port_num=$1;
if [ $# -lt 1 ]; then
    port_num=2100
fi

mount -t proc lxcproc /proc
/usr/sbin/sshd -D -p $port_num
umount /proc


$ ./pidns_exec ./lxc-wrap.sh

# Now from another window, login to the
# system where above sshd is running

$ ssh -p 2100 <user@hostname>

$ sleep 1000 &

$ ps -e -o"pid,ppid,pgid,sid,cmd"

 The pids shown should start at 1. Other processes
 in init_pid_ns should not be visible.

Test2: Ensure processes in child pid namespace are visible from
       init_pid_ns.

 From a third window, ssh into the test system on default port
 22 (i.e we should now be in init-pid-ns) and run:

 $ ps -e -o"pid,ppid,pgid,sid,cmd"

 All processes in the system, including the 'sleep 1000' process
 above should be visible.

 Note the different pids for the "sleep 1000" in Test1 and Test2.

Additional tests:

 Run a kernel build and/or LTP from the pid namespace created
 in Test1. (We ran kernel build. LTP TBD on current patchset).

Finally: reboot the system and ensure no crashes during shutdown :-)

---
/* begin pidns_exec.c */

/*
```

```
 * This code is trimmed-down version of Serge Hallyn's ns_exec.c
 * and focusses on testing cloning of pid namespace.
 */

/*
 * To create nested namespace:
 *
 *  pidns_exec -- lxc-wrap.sh
 *   In a pid namespace that is 1 level below init_pid_ns,
 *   execute lxc-wrap.sh
 *
 *  pidns_exec -N 3 -- lxc-wrap.sh
 *   In a pid namespace that is 3 levels below init_pid_ns,
 *   execute lxc-wrap.sh
 *
 * To run as fork bomb:
 *
 *  pidns_exec -F 32768 -- /bin/ls -l
 *
 *   Create 32768 processes that execute /bin/ls -l in
 *   current directory.
 *
 * Note that '--' is needed to separate args to pidns_exec from args to
 * command.
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <libgen.h>
#include <errno.h>
#include <sys/wait.h>

#ifndef SYS_unshare
#ifdef __NR_unshare
#define SYS_unshare __NR_unshare
#elif __i386__
#define SYS_unshare 310
#elif __ia64__
#define SYS_unshare 1296
#elif __x86_64__
#define SYS_unshare 272
#elif __s390x__
#define SYS_unshare 303
#elif __powerpc__
#define SYS_unshare 282
#else
#error "unshare not supported on this architecure."
```

```
#endif
#endif

#define CLONE_FS        0x00000200 /* set if fs info shared between processes */
#define CLONE_NEWNS     0x00020000      /* New namespace group? */
#define CLONE_NEWPID    0x20000000      /* New pid namespace */

static const char* procname;

enum test_mode {
 NESTED_NS,
 FORK_BOMB
};

static void usage(const char *name)
{
 printf("Usage:\n", name);
 printf("       %s [-N <num>] [command [arg ..]]\n", name);
 printf("       %s [-F <num>] [command [arg ..]]\n", name);
 printf("       %s [-h]\n", name);
 printf("\n\n");
 printf("  -F <num> (F)ork bomb <num> processes and have them run "
    "'command'\n\n");
 printf("  -N <num> Run 'command' in a pid ns (N)ested <num> "
    "levels below init_pid_ns\n\n");
 printf("  -h  this message\n");
 printf("\n");
 printf("If no options are specified, assume '-N 1'\n\n");
 printf("(C) Copyright IBM Corp. 2006\n");
 printf("\n");
 exit(1);
}

int do_exec(char *vargv[])
{
 char **argv = (char **)vargv;

 execve(argv[0], argv, __environ);
 perror("execve");
 exit(9);
}

do_fork_bomb(int num_procs, char *argv[])
{
 int i;
 int status;

 for (i = 0; i < num_procs; i++) {
```

```
  if (fork() == 0) {
   printf("%d: Fork-bomb: Ppid %d, Prog %s\n",
     getpid(), getppid(), (char *)argv[0]);
   do_exec(argv);
  }
 }

 while(wait(&status) != -1)
  ;

 _exit(0);
}

struct child_arg {
 char **argv;
 int  cur_depth;
 int  max_depth;
};

unshare_mnt_ns(struct child_arg *carg)
{
 /* To remount /proc in new pid ns, unshare mount ns */

 if (syscall(SYS_unshare, CLONE_NEWNS|CLONE_FS) < 0) {
  perror("unshare");
  exit(1);
 }
}

void *prepare_for_clone()
{
 void *stack;
 void *childstack;

 stack = malloc(getpagesize());
 if (!stack) {
  perror("malloc");
  exit(1);
 }
 childstack = stack + getpagesize();
 return childstack;
}

int do_nested_pid_ns(struct child_arg *carg)
{
 void *childstack;
 int ret;
 unsigned long flags = CLONE_NEWPID;
```

```
 int status;
 int pid;
 int rc;

 if (carg->cur_depth++ == carg->max_depth) {
  do_exec(carg->argv);
  return;
 }

 unshare_mnt_ns(carg);

 childstack = prepare_for_clone();

 ret = clone(do_nested_pid_ns, childstack, flags, (void *)carg);
 if (ret == -1) {
  perror("clone");
  return -1;
 }

 rc = waitpid(-1, &status, __WALL);
}

int main(int argc, char *argv[])
{
 int c;
 int status;
 struct child_arg carg;
 int rc;
 int mode = NESTED_NS;
 int num_procs = 32768;

 procname = basename(argv[0]);

 carg.max_depth = 1;
 carg.cur_depth = 0;

 while ((c = getopt(argc, argv, "F:hN:")) != EOF) {
  switch (c) {
  case 'F':
   mode = FORK_BOMB;
   num_procs = atoi(argv[optind-1]);
   break;
  case 'N':
   mode = NESTED_NS;
   carg.max_depth = atoi(argv[optind-1]);
   break;
  case 'h':
  default:
```

```
  usage(procname);
 }
};

carg.argv = &argv[optind];

if (mode == NESTED_NS) {
 printf("%d (%s): Creating nested pid ns of depth %d\n",
   getpid(), procname, carg.max_depth);
 do_nested_pid_ns(&carg);
} else {
 do_fork_bomb(num_procs, carg.argv);
}

while(waitpid(-1, &status, __WALL) != -1)
 ;

printf("%d (%s): Exiting\n", getpid(), procname);
exit(0);
}
```

_____

Containers mailing list
Containers@lists.linux-foundation.org
https://lists.linux-foundation.org/mailman/listinfo/containers