Subject: Summary: PID virtualization , Containers, Migration
Posted by Hubertus Franke on Fri, 03 Feb 2006 18:36:35 GMT
View Forum Message <> Reply to Message

Seems lots of interest is happening in this direction now
with various ideas and patches having been submitted.

I think the terms should be clarified a bit, even in our own
discussion we are using them loosely. Let's keep focusing
on the PID issues of containers and migration.

- user have come to expect that pids range from [0..PIDMAX]
- we can only migrate a self contained process tree (with some glue at the top)
- when we migrate we expect that the pid is preserved to the user
- we expect that UNIX semantics of pid visibility (e.g. kill etc) do
    not cross the container boundary (with some glue at the top)

Now there are two views to achieve the preservation that go at the heart
of the implementation only (not at the concept at all).
Two fundamental concepts are thrown around here:
(a) PID virtualization
(b) PID spaces

Let's be clear what I mean by these concepts:

PID virtualization:
===================
The kernel continues to use its current internal pid management.
Each task has a unique internal pid, throughout its life on that
particular kernel image.
At the user/kernel boundary these pids are virtualized on a
per container base. Hence pids coming from user space are converted to
an internal pid and internal pids handed to the user are converted
to the virtual pid.
As a result, all places along the user/kernel boundary that deal with
pids need to be identified and the conversion function needs to be called.
Also the vpid needs to be allocated/freed on fork/exit.

How this virtualization is maintained is an implementation detail.
One can keep a full "pid->vpid" and "vpi->pid" hash with the container
or play optimizations such as the OpenVZ folks to keep extra virtual
numbers in the task->pids[type] structure and or making
vpid == internal pid for non-containerized processes.

Right now the OpenVZ patch supplies a nice and clean means for this.
In a sense our patch also followes this approach because we explicitely
have these conversion functions and we too have similar optimizations
by masking the container id into the internal pid.

Other then on the details of the pid virtualization, the patches seem
to actually be the same ..

What has been pointed out is that
- calling these conversion function is overhead, cumbersome.
- confusing, hence dangerous, since we are maintaining different logical
  pid domains here under the same type pid_t

PID spaces:
===========

Pidspaces drive the isolation based on containers all the way down into the
current pid management. This at first sounds scary, but it will remove
lots of problem with the above approach.

Instead of keeping a single internal pid range and
its associated managment (aka pidspace), we provide one pidspace for each
container. All lookup functions operate on the container's pidspace,
hence this approach should remove the conversion functions and the two
logical domains, which seem to be big problem with the first approach.

Alan Cox made a very important statement, namely there is no reason that
we require unique pids in the kernel, even gave some examples where it isn't
true today. What becomes however a problem is the fact that now rather then
storing the pid as a reference, we need to now store the <pid,container> pair,
or more elegantly the pointer to the task. This later part unfortunately
has the problem of expired references to task structs.

I think that Eric's patch on weak task references, if I understood it correctly,
might come in handy. Instead of storing the task pointer, one stores the
task reference.

What needs to be solved is the container spawner transition into a new container
just like in the pid virtualization case.

Do people agree on these two characterizations and if so lets agree on what is
the right approach. From Linus's and Alan's comments I read they favor exploring
two, because of the problems listed with the first approach.

Other issues..

Herbert, brought up the issue of container hierarchy. Again, I think different
things come to mind here.

If indeed the containers are fully isolated from each other then its seems more a lifecycle management. E.g. migration of a container implies migration of all its children containers.
 From a kernel perspective, containers form a flat hierarchy.
 From a user's perspective one might allow certain operations (e.g. running the process list of another container) and here constraints can be enforced.

e.g.   ps --show-container mycontainer will show a specific containers