Why does running top inside docker container only show processes inside the container?

Asked 4 years ago Modified 4 years ago Viewed 2k times

2

I'm running top inside a docker container and I'm seeing that the only processes that show up are the initial process used to run the container and top. Why does it show this instead of displaying other processes on the docker host as well?

linux docker

Share Edit Follow

asked Apr 17, 2019 at 14:12

Peter Heo

23 ● 1 ● 4

1 Answer

Sorted by: Highest score (default)

7

In order to understand why this is happening, you need to understand the basic concepts of Linux that Docker is taking advantage of.

There is this feature in the Linux Kernel called `namespaces` that partitions/isolates the host resources in way where a set of processes sees one set of resources where as another set of processes sees another set of resources.

Linux has 7 types of namespaces:

- Mount - isolate mount points
- UTS - isolate hostname
- IPC - isolate interprocess communication resources
- PID - isolate the PID number space

- Network - isolate network interfaces
- User - isolate UID/GID number spaces
- Cgroup - isolate cgroup root directory

When you are working on your linux machine everything that you do is on the same namespace, but when you create a container by doing `docker run` by default it'll create a new separate namespace to isolate the container from your host.

In the specific case of your question, you see just one process running because the container is in a different PID namespace as your host machine.

You can tell Docker to share the same PID namespace by using `--pid="host"` when you create the container, there are some cases when doing that is useful.

Share Edit Follow

answered Apr 17, 2019 at 14:36

Esteban Garcia
2,151 ● 16 ● 24

The Most Pointless Docker Command Ever

2 Minutes

What?

This article will show you how you can undo the things Docker does for you in a Docker command. Clearer now?

OK, Docker relies on Linux namespaces to isolate effectively copy parts of the system so it ends up looking like you are on a separate machine.

For example, when you run a Docker container:

```
$ docker run -ti busybox ps -a
PID USER COMMAND
 1 root ps -a
```

it only 'sees' its own process IDs. This is because it has its own PID namespace.

Similarly, you have your own network namespace:

```
$ docker run -ti busybox netstat -a
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address Foreign Address State
Active UNIX domain sockets (servers and established)
Proto RefCnt Flags Type State I-Node Path
```

You also have your own view of inter-process communication and the filesystem.

Go on then

This is possibly the most pointless possible docker command ever run, but here goes:

```
docker run -ti
    --privileged
    --net=host --pid=host --ipc=host
    --volume /:/host
    busybox
    chroot /host
```

The three '=host' flags bypass the network, pid and ipc namespaces. The volume flag mounts the root filesystem of the host to the '/host' folder in the container (you can't mount to '/' in the container). The privileged flags gives the user full access to the root user's capabilities.

All we need is the chroot command, so we use a small image (busybox) to chroot to the filesystem we mounted.

What we end up with is a Docker container that is running as root with full capabilities in the host's filesystem, will full access to the network, process table and IPC constructs on the host. You can even 'su' to other users on the host.

If you can think of a legitimate use for this, please drop me a line!

Why?

Because you can!

Also, it's quite instructive. And starting from this, you can imagine scenarios where you end up with something quite useful.

Imagine you have an image – called 'filecheck' – that runs a check on the filesystem for problematic files. Then you could run a command like this (which won't work BTW – filecheck does not exist):

```
docker run --workdir /host -v /:/host:ro filecheck
```

This modified version of the pointless command dispenses with the chroot in favour of changing the workdir to '/host', and – crucially – the mount now uses the ':ro' suffix to mount the host's filesystem read-only, preventing the image from doing damage to it.

So you can check your host's filesystem relatively safely without installing anything.

You can imagine similar network or process checkers running for their namespaces.

Can you think of any other uses for modifications of this pointless command?
This post is based on material from Docker in Practice (http://manning.com/miell/?a_aid=zwischenzugs&a_bid=e0d48f62), available on Manning's Early Access Program. ***Get 39% off with the code: 39miell***