LINUX CONTAINERS IN A FEW LINES OF CODE

This post is meant to be a counterpart to the previous KVM post, but about containers instead. The idea is to show how exactly containers work by running a busybox Docker image in our own small container runner.

Unlike a VM, container is very vague term. Normally, we call containers a self-contained bundle of code and its dependencies, that can be shipped together and run in an isolated environment inside some host operating system. If it still sounds similar to a VM, let's dive deeper and see how containers are implemented.

BUSYBOX DOCKER

Our end goal would be to run a common busybox Docker image, but without docker. Docker uses btrfs as a filesystem format for its images. Let's try pulling the image and unpack it into a directory:

```
mkdir rootfs
docker export $(docker create busybox) | tar -C rootfs -xvf -
```

Now we got busybox image filesystem unpacked into the `rootfs` folder. Surely, we can run `./rootfs/bin/sh` and get a working shell, but if we look at the list of the processes there, or files, or network interfaces - we will see that we have access to our whole OS.

So, let's try to create an isolated environment somehow.

CLONE

Since we want to control what a child process can see, we will be using clone(2) instead of fork(2). Clone does pretty much the same, but allows you to pass flags, defining which resources you would want to share.

The following flags are allowed:

- `CLONE_NEWNET` – isolate network devices
- `CLONE_NEWUTS` – host and domain names (UNIX Timesharing System)
- `CLONE_NEWIPC` – IPC objects
- `CLONE_NEWPID` – PIDs
- `CLONE_NEWNS` – mount points (file systems)
- `CLONE_NEWUSER` – users and groups

In our experiment we will try to isolate processes, IPC, network and file systems, so here we go:

```
static char child_stack[1024 * 1024];

int child_main(void *arg) {
  printf("Hello from child! PID=%d\n", getpid());
  return 0;
}

int main(int argc, char *argv[]) {
  int flags =
      CLONE_NEWNS | CLONE_NEWUTS | CLONE_NEWPID | CLONE_NEWIPC | CLONE_NEWNET;
  int pid = clone(child_main, child_stack + sizeof(child_stack),
                  flags | SIGCHLD, argv + 1);
  if (pid < 0) {
    fprintf(stderr, "clone failed: %d\n", errno);
    return 1;
  }
  waitpid(pid, NULL, 0);
  return 0;
}
```

It has be run with superuser privileges, otherwise clone would fail.

This gives an already interesting result: child PID is 1. We all know that PID 1 is normally `init`, but in this case we see that child process got its own isolated list of processes, where it has become the first process.

RUNNING SHELL

To easier play around the new environment, let's run a shell in the child process. Actually, let's run arbitrary commands, much like `docker run`:

```
int child_main(void *arg) {
  char **argv = (char **)arg;
  execvp(argv[0], argv);
  return 0;
}
```

Now, running our app with "/bin/sh" argument opens a real shell, where we can type commands. This shows how wrong we were about the isolation:

```
# echo $$
1
# ps
  PID TTY          TIME CMD
 5998 pts/31   00:00:00 sudo
 5999 pts/31   00:00:00 main
 6001 pts/31   00:00:00 sh
 6004 pts/31   00:00:00 ps
```

As we see, the shell process itself has a PID of 1, but actually can see and access all the other processes from the host OS. The reason is that the list of processes is read from procfs, which is still inherited.

So, let's unmount procfs:

```
umount2("/proc", MNT_DETACH);
```

Now, running the shell breaks ps, mount, and other command, because no procfs is mounted. Still better than leaking the parent's procfs.

CHROOT

In the old days chroot was a "good enough" isolation for most use cases, but here let's use a `pivot_root` instead. This system call moves and existing rootfs into some subdirectory, and make a another directory a new root:

```
int child_main(void *arg) {
  /* Unmount procfs */
  umount2("/proc", MNT_DETACH);
  /* Pivot root */
  mount("./rootfs", "./rootfs", "bind", MS_BIND | MS_REC, "");
  mkdir("./rootfs/oldrootfs", 0755);
  syscall(SYS_pivot_root, "./rootfs", "./rootfs/oldrootfs");
  chdir("/");
  umount2("/oldrootfs", MNT_DETACH);
  rmdir("/oldrootfs");
  /* Re-mount procfs */
  mount("proc", "/proc", "proc", 0, NULL);
  /* Run the process */
  char **argv = (char **)arg;
  execvp(argv[0], argv);
  return 0;
}
```

It would make sense to mount tmpfs into /tmp, sysfs into /sys and create a valid /dev filesystem, but to keep things short I'll skip it.

Anyway, now we only see the files from busybox image rootfs, as if we chrooted into it:

```
/ # ls
bin   dev   etc   home  proc  root  sys   tmp   usr   var

/ # mount
/dev/sda2 on / type ext4 (rw,relatime,data=ordered)
proc on /proc type proc (rw,relatime)

/ # ps
PID   USER     TIME  COMMAND
    1 root      0:00 /bin/sh
    4 root      0:00 ps

/ # ps ax
PID   USER     TIME  COMMAND
    1 root      0:00 /bin/sh
    5 root      0:00 ps ax
```

At this point it looks more or less isolated, perhaps, too isolated - we can not ping anything and the network does not seem to work at all.

NETWORK

Creating a new network namespace was only the start. We need to assign network interfaces to it and set them up to do proper packet forwarding.

If you don't have a br0 interface, let's create it manually (brctl is part of bridge-utils package on Ubuntu):

```
brctl addbr br0
ip addr add dev br0 172.16.0.100/24
ip link set br0 up
sudo iptables -A FORWARD -i wlp3s0  -o br0 -j ACCEPT
sudo iptables -A FORWARD -o wlp3s0 -i br0 -j ACCEPT
sudo iptables -t nat -A POSTROUTING -s 172.16.0.0/16 -j MASQUERADE
```

In my case, wlp3s0 was my primary WiFi network interface and 172.16.x.x was a network for the container.

What our container launcher should do is to create a pair of peer interfaces, veth0 and veth1, link them to the br0 and set up routing within the container.

In the main() function we will run these commands before clone:

```
system("ip link add veth0 type veth peer name veth1");
system("ip link set veth0 up");
system("brctl addif br0 veth0");
```

After clone() is complete, we will add veth1 to the new child namespace:

```
char ip_link_set[4096];
snprintf(ip_link_set, sizeof(ip_link_set) - 1, "ip link set veth1 netns %d",
         pid);
system(ip_link_set);
```

Now if we run “ip link” in the container shell we will see a loopback interface, and some veth1@xxxx interface. But the network still doesn’t work. Let’s set a unique hostname in the container and configure the routes:

```
int child_main(void *arg) {

  ....

  sethostname("example", 7);
  system("ip link set veth1 up");

  char ip_addr_add[4096];
  snprintf(ip_addr_add, sizeof(ip_addr_add),
           "ip addr add 172.16.0.101/24 dev veth1");
  system(ip_addr_add);
  system("route add default gw 172.16.0.100 veth1");

  char **argv = (char **)arg;
  execvp(argv[0], argv);
  return 0;
}
```

Let’s see how it looks like:

```
/ # ip link
1: lo: <LOOPBACK> mtu 65536 qdisc noop qlen 1
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
47: veth1@if48: <BROADCAST,MULTICAST,UP,LOWER_UP,M-DOWN> mtu 1500 qdisc noqueue qlen 1
    link/ether 72:0a:f0:91:d5:11 brd ff:ff:ff:ff:ff:ff

/ # hostname
example

/ # ping 1.1.1.1
PING 1.1.1.1 (1.1.1.1): 56 data bytes
64 bytes from 1.1.1.1: seq=0 ttl=57 time=27.161 ms
64 bytes from 1.1.1.1: seq=1 ttl=57 time=26.048 ms
64 bytes from 1.1.1.1: seq=2 ttl=57 time=26.980 ms
...
```

It works!

SUMMARY

The full source code is available at https://gist.github.com/zserge/4ce3c1ca837b96d58cc5bdcf8befb80e. If you found a mistake or got a suggestion - please leave a comment there!

Obviously, Docker does much more than this. But it's amazing how many convenient APIs Linux kernel has and easy it is to use them to achieve OS-level virtualization.

I hope you've enjoyed this article. You can follow – and contribute to – on Github, Twitter or subscribe via rss.

May 10, 2020

See also: KVM host in a few lines of code and more.