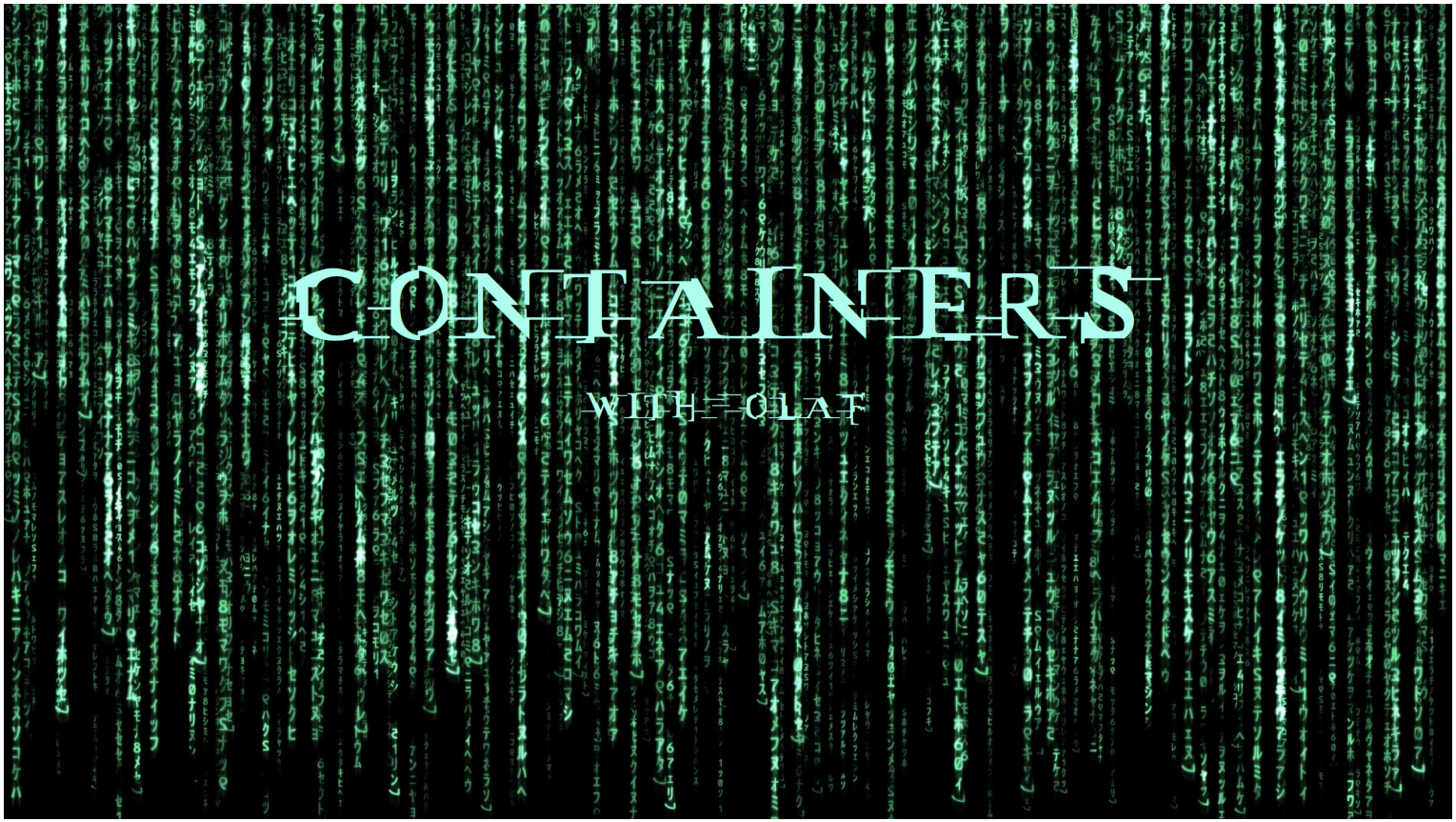

CONTAINERS
WITH OLAF

You want to know what containers are?

The Spoon

- Do not try to run containers, that's impossible. Instead, only try to realize the truth...
- What truth?
- There is no container...
- There is no container?
- Then you'll see that it is not the container which runs, it is the process itself.

What Is A Process?

- it has its own private memory
- violations against process memory borders get a SIGSEGV(11)
- a process has a heap, a stack, code (TEXT) and data (ANON)
- the process can be observed by ps(1), which shows some attributes:

```
$ ps -fp $$
     UID   PID  PPID   C    STIME TTY          TIME CMD
olbohlen 11651 10046   0 23:07:43 pts/6       0:00 ksh
```

- we see the user id, process id, parent-pid, start time, the tty, the cpu time and command name
- in UNIX these attributes are bundled in a C structure called proc_t
- Linux uses task_struct which is a more hierarchical structure

Container Implementations

There are various implementations:

- Linux: OpenVZ (2005), docker (2013), podman (~2018), etc. . .
- FreeBSD: jails (Mar 2000)
- illumos/Solaris: containers (Feb 2004)
- AIX: wpars

and various others. . .

Welcome to a training program, let's start a simple container with podman...

```
[olbohlen@rhel85 ~]$ podman run -d ubi8 sleep 10000
6b336fb0012f6f3d8fadca333e1e2bd900b7ede9560594bb0c5acc27a3aef4ee
[olbohlen@rhel85 ~]$ podman ps
CONTAINER ID  IMAGE                                       COMMAND      CREATED         STATUS
6b336fb0012f  registry.access.redhat.com/ubi8:latest  sleep 10000  2 seconds ago  Up 2 seconds
[olbohlen@rhel85 ~]$ ps -ef | grep "sleep 10000"
olbohlen    5026    5017  0 23:44 ?        00:00:00 /usr/bin/coreutils --coreutils-prog-shebang
[root@6b336fb0012f /]# ps -ef | grep sleep
root           1       0  0 22:44 ?        00:00:00 /usr/bin/coreutils --coreutils-prog-shebang
[olbohlen@rhel85 ~]$ ps -fZp 5017
LABEL                             UID          PID    PPID  C STIME TTY          TIME CMD
unconfined_u:system_r:container_runtime_t:s0 olbohlen 5017 1  0 23:44 ?    00:00:00 /usr/bin/
```

- podman uses runc(8) - the OCI container runtime
- containers are instantiated using different technologies
  - namespaces: providing resource "visibilities"
  - cgroups: limiting compute resources as cpu and memory
  - chroot: creating a fake root directory
  - seccomp: limiting access to systemcalls
  - SELinux: proving extra layers to prevent escapes

Namespaces "scope" the visibility of various things Linux supports different types of namespaces(7) like:

- cgroup: Cgroup root directory
- ipc: System V IPC, POSIX message queues
- mnt: Mount points
- net: Network devices, stacks, ports, etc.
- pid: Process IDs
- user: User and group IDs
- uts: Hostname and NIS domain name

Which can isolate processes in different ways Namespaces can be created by unshare(1)

Let's build a simple container on our own with unshare(1) and chroot(1):

```
$ mkdir -p ~/sysroot/{bin,lib64,proc}
$ for f in $(ldd /bin/{bash,df,ls,lsns,mount,ps,uname} | \
> tr '[       :]' '\n' | grep /); do cp $f sysroot/$f; done
$ sudo mount --bind /home/olbohlen/sysroot/proc /home/olbohlen/sysroot/proc
$ unshare -irmnpuUCf --mount-proc=$PWD/sysroot/proc chroot $PWD/sysroot /bin/bash
bash-4.4# /bin/ps -ef
UID          PID    PPID  C STIME TTY          TIME CMD
0              1       0  0 16:58 ?        00:00:00 /bin/bash
0              2       1  0 16:58 ?        00:00:00 /bin/ps -ef
bash-4.4# /bin/mount
/dev/mapper/rhel_rhel85-root on /proc type xfs (rw,relatime,seclabel,attr2,inode64,logbufs=8,
proc on /proc type proc (rw,nosuid,nodev,noexec,relatime)
```

Well, we have to trust Linux here a bit... But on other UNIX systems we can actually dig deeper:
Our rabbit hole entry is the kernel debugger, which we can attach to a running UNIX kernel and observe (and modify) the system live.
Allow me to do that on illumos, as the process structures are a bit more "organized".

When we attach the kernel debugger (mdb) against a running kernel, we have raw memory access. UNIX organizes data in C structures, which may contain other data types such as int or char (or again structs).
A simple C structure could look like this:

```
struct position {
  int x;
  int y;
};
```

And if we would read the struct it may look like:

```
position.x = 42
position.y = 23
```

(Un)fortunately the debugger does not know the format of a data structure at a given address, so we need to validate that we got correct data.
The debugger has some commands to look at known places for certain structures, such as the process table or in our example the list of containers.

So let's run the debugger:

```
(701) x230:/root# mdb -k
Loading modules: [ unix genunix specfs dtrace mac cpu.generic uppc apix scsi_vhci zfs sata sd
> ::zone
            ADDR     ID STATUS         NAME                 PATH
ffffffffbd08c20      0 running        global               /
fffffe16886626c0     1 running        asterix              /export/zones/asterix/root/
fffffe16929ebd80     2 running        obelix               /export/zones/obelix/root/
fffffe16bbc66500     4 running        rhel85               /export/zones/rhel85/root/
```

Wait, we have a container called "rhel85", wasn't that the rhel machine from the demos before? Yes, actually that container runs a bhyve hypervisor process which runs RHEL 8.5...

```
(629) x230:/export/home/olbohlen$ ps -f -z rhel85
     UID   PID  PPID   C    STIME TTY         TIME CMD
    root 15267  5136   0 17:57:03 ?           1:40 /usr/sbin/bhyve -U 37960a3a-c5ac-6c8b-d14b
    root  5113     1   0   Jan 01 ?           0:00 zsched
    root  5136  5113   0   Jan 01 ?           0:00 /usr/bin/python3.5 /usr/lib/brand/bhyve/in
(630) x230:/export/home/olbohlen$
```

We have the bhyve process with the PID 15267 running according to ps(1), let's look in mdb:

```
> ::ps ! egrep "(PID|bhyve)"
S PID    PPID   PGID   SID    UID    FLAGS      ADDR             NAME
R 15267  5136   5113   5113   0      0x4a004000 fffffe16b8d6d010 bhyve
```

ADDR is the start address in RAM for the proc_t data structure

```
> fffffe16b8d6d010::print -a proc_t ! less
[...]
> fffffe16b8d6d010::print -a proc_t p_user.u_psargs
fffffe16b8d6d879 p_user.u_psargs = [ "/usr/sbin/bhyve -U 37960a3a-c5ac-6c8b-d14b-8204ca044474
```

The proc_t structure store all attributes to a process, so those that ps(1) shows and more. Also in that proc_t we have the container id in it (p_zone, think of it as the namespace id):

```
> fffffe16b8d6d010::print -a proc_t p_zone
fffffe16b8d6d658 p_zone = 0xfffffe16bbc66500
> 0xfffffe16bbc66500::zone
            ADDR     ID STATUS        NAME                 PATH
fffffe16bbc66500      4 running       rhel85               /export/zones/rhel85/root/
>
```

- We need images. Lots of images.

podman/docker use so called images to instantiate containers. These images are made of Layers, like viewfoils on overhead projectors.

```
[olbohlen@rhel85 scratch]$ skopeo inspect docker://registry.access.redhat.com/rhscl/postgresq
> | jq ".Layers"
[
  "sha256:ac08ca107ad9ed699cbd28339749dd6463a84c73aa1d468a4241385fc4ec3876",
  "sha256:b46ca46c303b49d886a7585735ebd1dc8651e83d0fab5823300cf3a9fd2febc1",
  "sha256:cdd22b43a6f986fc909d504043ef6ad6528a6c1927f27c80eea2d19ffe5079fe",
  "sha256:4c9f611df095eef49c081f758ad314b62a297172e22a8a746514d252a7a89c45"
]
```

This image contains four layers which itself are tar archives which you can extract.

Let's extract an image to a local directory:

```
[olbohlen@rhel85 scratch]$ skopeo copy --remove-signatures \
> docker://registry.access.redhat.com/rhscl/postgresql-10-rhel7 dir:///$PWD
Copying blob ac08ca107ad9 done
Copying blob b46ca46c303b done
Copying blob cdd22b43a6f9 done
Copying blob 4c9f611df095 done
Copying config 00a55534f8 done
Writing manifest to image destination
Storing signatures
[olbohlen@rhel85 scratch]$ ls
00a55534f8db45877d6657cc9b1ba77841c49cb21cc4d7a4c9cd4e98020a4bc8
4c9f611df095eef49c081f758ad314b62a297172e22a8a746514d252a7a89c45
ac08ca107ad9ed699cbd28339749dd6463a84c73aa1d468a4241385fc4ec3876
b46ca46c303b49d886a7585735ebd1dc8651e83d0fab5823300cf3a9fd2febc1
cdd22b43a6f986fc909d504043ef6ad6528a6c1927f27c80eea2d19ffe5079fe
manifest.json
version
```

Also use jq(1) to inspect the manifest and the config.

Finding The Image Config

There's an obvious manifest.json, so let's look into it.

```
[olbohlen@rhel85 scratch]$ jq ".config.digest" <manifest.json
"sha256:00a55534f8db45877d6657cc9b1ba77841c49cb21cc4d7a4c9cd4e98020a4bc8"
```

That's our image config, itself a json file:

```
[olbohlen@rhel85 scratch]$ jq . 00a55534f8db45877d6657cc9b1ba77841c49cb21cc4d7a4c9cd4e98020a4
{
  "architecture": "amd64",
[...]
```

Looks familiar? Yes, that's more or less podman inspect.

In the manifest.json we also see the layers:

```
[olbohlen@rhel85 scratch]$ jq ".layers[].digest" manifest.json
"sha256:ac08ca107ad9ed699cbd28339749dd6463a84c73aa1d468a4241385fc4ec3876"
"sha256:b46ca46c303b49d886a7585735ebd1dc8651e83d0fab5823300cf3a9fd2febc1"
"sha256:cdd22b43a6f986fc909d504043ef6ad6528a6c1927f27c80eea2d19ffe5079fe"
"sha256:4c9f611df095eef49c081f758ad314b62a297172e22a8a746514d252a7a89c45"
[olbohlen@rhel85 scratch]$ du -h ac08ca107ad9ed699cbd2833[...]
73M     ac08ca107ad9ed699cbd28339749dd6463a84c73aa1d468a4241385fc4ec3876
4.0K    b46ca46c303b49d886a7585735ebd1dc8651e83d0fab5823300cf3a9fd2febc1
7.0M    cdd22b43a6f986fc909d504043ef6ad6528a6c1927f27c80eea2d19ffe5079fe
33M     4c9f611df095eef49c081f758ad314b62a297172e22a8a746514d252a7a89c45
```

These are tar(1) archives we can extract and inspect. When you start a container, the extracted layers will be mounted with OverlayFS.

podman uses fuse-overlayfs(1) to mount container image layers.
Since Linux 4.18 this can be done also by non-root users:

```
$ mkdir layer1
$ mkdir layer2
$ mkdir ephemeral-layer
$ mkdir mountdir
$ echo "this is file one" >layer1/f1
$ echo "this is file two" >layer2/f2
$ fuse-overlayfs -o lowerdir=$PWD/layer1:$PWD/layer2 -o upperdir=$PWD/ephemeral-layer \
> -o workdir=$PWD/fuse-work $PWD/mountdir
$ ls mountdir
f1  f2
$ echo "this is file three" >mountdir/f3
$ fusermount -u $PWD/mountdir
$ ls */f?
ephemeral-layer/f3  layer1/f1  layer2/f2
```

- We need an exit!

podman uses CNI (Container Native Interface) to provide a network interface for a container (so, a namespaced NIC), which will be usuall created on a bridge. This is only possible for containers started as root:

```
[olbohlen@rhel85 ~]$ sudo podman run -it registry.access.redhat.com/ubi8 \
> bash -c "(dnf install -y iproute && ip a s)"
Updating Subscription Management repositories.
[...]
Complete!
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: eth0@if7: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    link/ether ca:dc:cc:3a:c9:e5 brd ff:ff:ff:ff:ff:ff link-netnsid 0
    inet 10.88.0.3/16 brd 10.88.255.255 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::c8dc:ccff:fe3a:c9e5/64 scope link
       valid_lft forever preferred_lft forever
```

Since a normal user can't instantiate interfaces usually, rootless containers can't use an interface on a bridge. Instead rootless containers use the userland tap driver (known from openvpn or virtualbox for example):

```
[olbohlen@rhel85 ~]$ podman run -it registry.access.redhat.com/ubi8 \
> bash -c "(dnf install -y iproute && ip a s)"
Updating Subscription Management repositories.
[...]
Complete!
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: tap0: <BROADCAST,UP,LOWER_UP> mtu 65520 qdisc fq_codel state UNKNOWN group default qlen 10
    link/ether 86:21:df:f9:40:43 brd ff:ff:ff:ff:ff:ff
    inet 10.0.2.100/24 brd 10.0.2.255 scope global tap0
       valid_lft forever preferred_lft forever
    inet6 fe80::8421:dfff:fef9:4043/64 scope link
       valid_lft forever preferred_lft forever
```

The tap driver is part of the universal tun/tap driver being developed since 1999 for Linux, FreeBSD and Solaris. It allows user processes to create an interface. Depending on your code it will create a tun or a tap interface.
What is the difference?

- a tun interface behaves like a Point-To-Point interface and handles IP packets
- a tap interface behaves like a Ethernet interface and handles Ethernet frames

All packets sent to these interfaces will be received by the application which created them. Popular examples are the pppd(8) or openvpn.
podman uses slirp4netns(1) to create a user-mode network interface

so, first we set up our simple container again:

```
$ mkdir -p ~/sysroot/{bin,lib64,proc,sbin}
$ for f in $(ldd /bin/{bash,df,ls,lsns,mount,ps,uname,ping} /sbin/{ip,ifconfig} | \
> tr '[       :]' '\n' | grep /); do cp $f sysroot/$f; done
$ sudo mount --bind /home/olbohlen/sysroot/proc /home/olbohlen/sysroot/proc
$ unshare -irmnpuUCf --mount-proc=$PWD/sysroot/proc chroot $PWD/sysroot /bin/bash
bash-4.4# /bin/ps -ef
UID          PID    PPID  C STIME TTY          TIME CMD
0              1       0  0 16:58 ?        00:00:00 /bin/bash
0              2       1  0 16:58 ?        00:00:00 /bin/ps -ef
bash-4.4# /bin/mount
/dev/mapper/rhel_rhel85-root on /proc type xfs (rw,relatime,seclabel,attr2,inode64,logbufs=8,
proc on /proc type proc (rw,nosuid,nodev,noexec,relatime)
```

On the host OS:

```
[olbohlen@rhel85 ~]$ pgrep -P $(pgrep -x unshare) bash
2425
[olbohlen@rhel85 ~]$ slirp4netns --configure --mtu=65520 2425 tap0
sent tapfd=5 for tap0
received tapfd=5
Starting slirp
 * MTU:             65520
 * Network:         10.0.2.0
 * Netmask:         255.255.255.0
 * Gateway:         10.0.2.2
 * DNS:             10.0.2.3
 * Recommended IP:  10.0.2.100
WARNING: 127.0.0.1:* on the host is accessible as 10.0.2.2 (set --disable-host-loopback to pr
```

Back in the container:

```
bash-4.4# /sbin/ip a s
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: tap0: <BROADCAST,UP,LOWER_UP> mtu 65520 qdisc fq_codel state UNKNOWN group default qlen 10
    link/ether be:0c:f2:d0:28:79 brd ff:ff:ff:ff:ff:ff
    inet 10.0.2.100/24 brd 10.0.2.255 scope global tap0
       valid_lft forever preferred_lft forever
    inet6 fe80::bc0c:f2ff:fed0:2879/64 scope link
       valid_lft forever preferred_lft forever
bash-4.4# /bin/ping 10.0.2.2
PING 10.0.2.2 (10.0.2.2) 56(84) bytes of data.
64 bytes from 10.0.2.2: icmp_seq=1 ttl=255 time=0.563 ms
64 bytes from 10.0.2.2: icmp_seq=2 ttl=255 time=0.127 ms
^C
--- 10.0.2.2 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1007ms
rtt min/avg/max/mdev = 0.127/0.345/0.563/0.218 ms
```

A bigger issue is that a user cannot start processes with different uids. For podman rootless containers, there is a UID mapping. The file /etc/subuid specifies a range of uids per user:

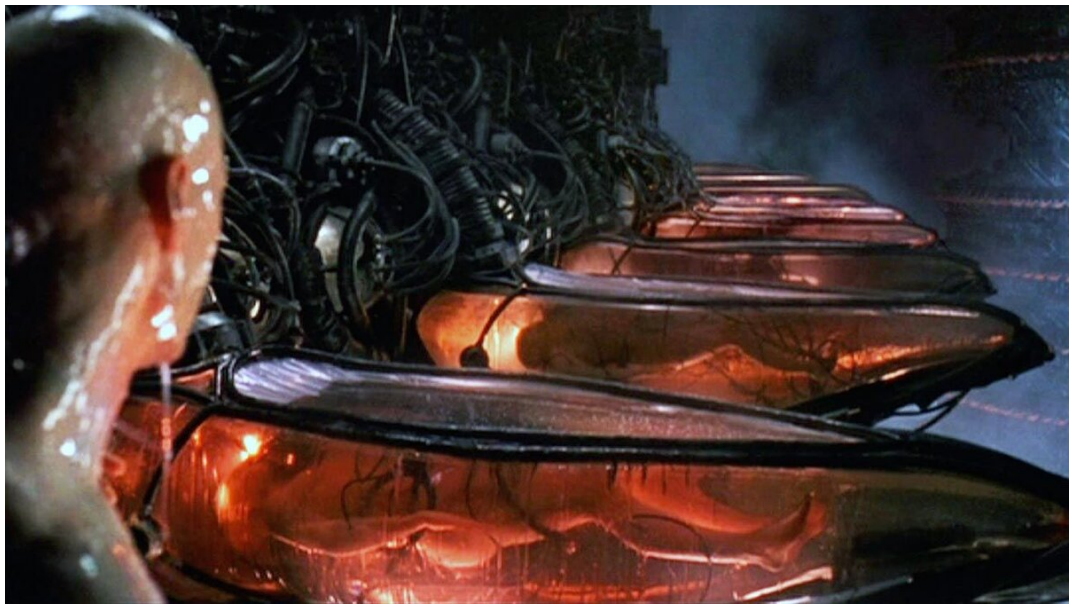
```
[olbohlen@rhel85 ~]$ id -a
uid=4100(olbohlen) gid=4100(olbohlen) groups=4100(olbohlen),10(wheel) context=unconfined_u:unc
[olbohlen@rhel85 ~]$ cat /etc/subuid
olbohlen:100000:65536
```

That means all uids from 100000 to 165535 are reserved for olbohlen. The mapping looks like this:

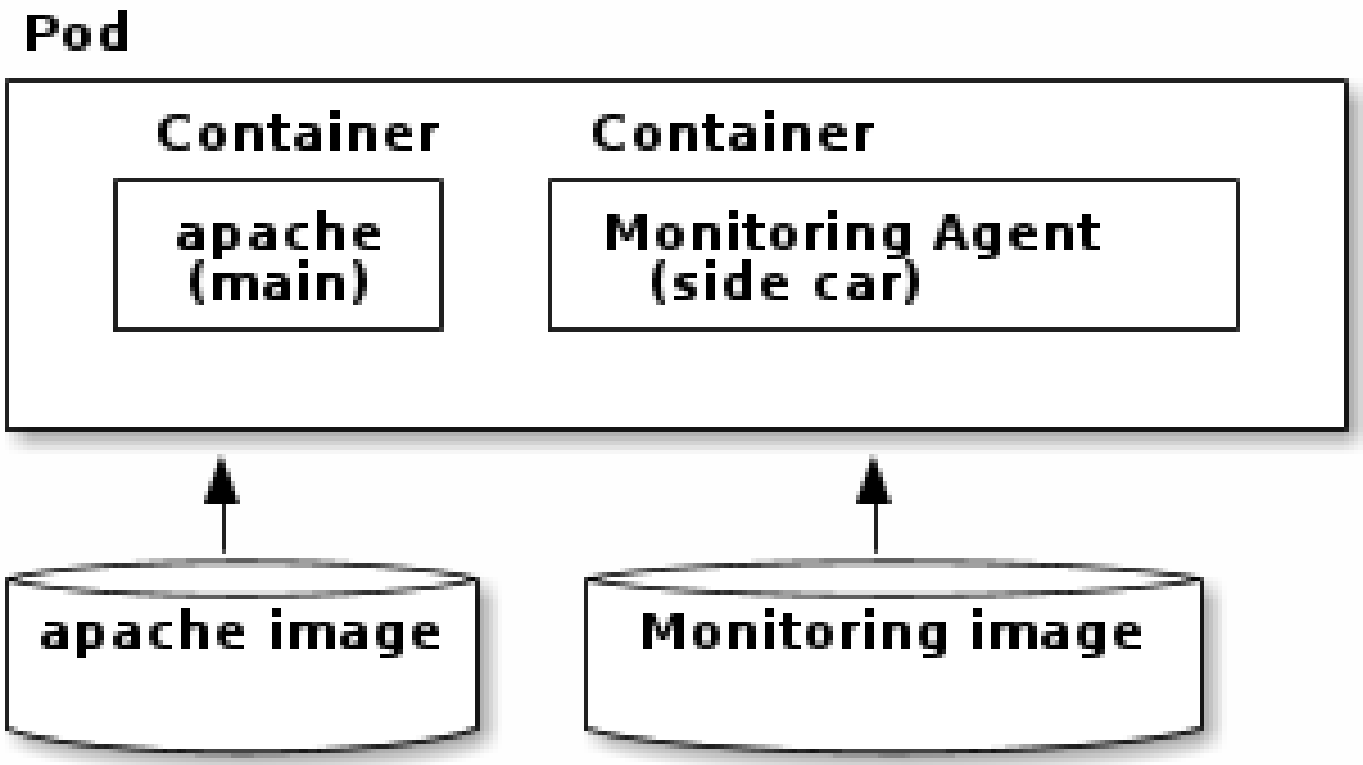
uid in container	uid outside container
0	4100 (users primary uid)
1	100000 (first subuid)
2	100001
...	...

And What Is A Pod?

- a pod is a set of containers
- usually contains side-car containers
- these containers share certain namespaces



Kubernetes does not manage containers, it manages pods as the most atomic item.



The idea is to seperate applications from helper applications to provide seperate releases.

So, what namespaces does a pod share between containers?

- net: They share the IP address and ports
- ipc: so you can use IPC (shared memory, semaphores, etc)
- uts: all containers share the same hostname

You can also enable sharing the PID namespaces by setting:
v1.pod.spec.shareProcessNamespace: true

```
(738) x230:/export/home/olbohlen/scratch$ oc logs hi-7459f5c556-qkxj4
error: a container name must be specified for pod hi-7459f5c556-qkxj4,
choose one of: [hi sidecarone]
(741) x230:/export/home/olbohlen/scratch$ oc rsh -c sidecarone hi-7459f5c556-qkxj4 ps -ef
PID   USER     TIME  COMMAND
    1 10006000  0:00 sleep 360000
    9 10006000  0:00 ps -ef
(747) x230:/export/home/olbohlen/scratch$ oc rsh -c hi hi-7459f5c556-qkxj4 ps -ef
UID          PID    PPID  C STIME TTY          TIME CMD
1000600+       1       0  0 19:33 ?        00:00:00 httpd -D FOREGROUND
1000600+      26       1  0 19:33 ?        00:00:00 /usr/bin/coreutils --coreuti
1000600+      27       1  0 19:33 ?        00:00:00 /usr/bin/coreutils --coreuti
1000600+      28       1  0 19:33 ?        00:00:00 /usr/bin/coreutils --coreuti
1000600+      29       1  0 19:33 ?        00:00:00 /usr/bin/coreutils --coreuti
1000600+      30       1  0 19:33 ?        00:00:00 httpd -D FOREGROUND
1000600+      36       1  0 19:33 ?        00:00:00 httpd -D FOREGROUND
1000600+      43       1  0 19:33 ?        00:00:00 httpd -D FOREGROUND
1000600+      64       1  0 19:33 ?        00:00:00 httpd -D FOREGROUND
1000600+      66       1  0 19:33 ?        00:00:00 httpd -D FOREGROUND
1000600+      72       1  0 19:33 ?        00:00:00 httpd -D FOREGROUND
1000600+      82       1  0 19:33 ?        00:00:00 httpd -D FOREGROUND
1000600+      88       1  0 19:33 ?        00:00:00 httpd -D FOREGROUND
1000600+     106       0  0 19:42 pts/0    00:00:00 ps -ef
```

Thank you for your attention.
Do you have any questions?
Feel free to ask now or contact me later:
olaf.bohlen@niit.com