OPENSOURCE CONTAINER

PODMAN

BUILDAH

SKOPEO

최국현

tang@linux.com

기반기술

역사 + 이론

RUNTIME HISTORY

KUBERNETES HISTORY

LOW AND HIGH LEVEL

기반기술

1. SELinux
2. Linux Capabilities
3. SECCOMP
4. Namespace(USER)

SELINUX

SELINUX

SELinux는 필수적인 컨테이너 구성원은 아니다.

하지만, SELinux를 사용하는 경우 좀 더 안정적으로 운영이 가능하다. SELinux는 다음과 같은 대표적인 구성 요소로 되어있다.

- fcontext
- port
- boolean

SECCOMP

SECCOMP는 필수 구성원이다. 특정 시스템 콜을 SECCOMP그룹에 넣어서 콜을 제한한다. 이를 통해서 프로세스를 통해서 위험한 시스템 콜을 차단 할 수 있도록 한다.

SECCOMP

process
open
read
execve
...
set seccomp mode
task_struct
make system calls
filter mode
filters
filter1
filter2
filter3
SIGKILL
system call in kernel

NAMESPACE

네임스페이스는 가상화 기술 중 하나이다.

이 기술은 본래 리눅스 'vServer Project'의 결과물 중 하나다. 네임스페이스는 기능은 프로세스에서 사용하는 각기 다른 자원들을 NAMESPACE(이하 NS)영역으로 분리 및 격리한다.

이를 통해서 얻는 장점은 오버헤드를 최소화 한다.

A. 네트워크 장치를 네임스페이스를 통해서 가상화 장치로 제공한다.

B. 파일 시스템에 마운트가 되는 부분을 네임스페이스를 통해서 컨테이너에 제공한다.

C. Pid 네임스페이스는 컨테이너에서 동작하는 프로세스가 호스트 영역의 프로세스 정보를 확인 할 수 없도록 한다.

NAMESPACE

이 기술은 유닉스 계열 시스템에서는 하드웨어 혹은 논리적으로 1970년도에 구현이 되었다.

리눅스 시스템은 2001, 2004, 2006년도 SELinux, namespace, cgroup를 통해서 구현이 되었다.

NAMESPACE

KERNEL
NAMESPACE
kernel side
driver
user side
driver
USER
NAMESPACE
NAMESPACE ID
APP
APP

NAMESPACE

KERNEL NAMESPACE
cgroup
USER NAMESPACE
MEMORY
CPU
USER PROGRAM
DISK
NET
UTS

NAMESPACE

USERSPACE
KERNEL SPACE
BASH
GETPID()
pid
uid/gid
net
STORAGE SUBSYSTEM
cat /etc/hostname
OPEN()
VFS
XFS
BLOCK

NAMESPACE

APPLICATION CODE
C LIBRARY
USER SPACE
SYSTEM INTERFACE
GENERIC SERVICES
DEVICES DRIVER
KERNEL SPACE
HARDWARE

NAMESPACE

Mount NS 1
Mount NS 2
/
X
Z
Y
/
X'
Y'
bind mount
Replicated into same peer
group during NS creation

컨테이너 구조

명령어로 컨테이너 기능 구현

2023-04-17

UNSHARE(ipc)

UNSHARE(uts)

readlink, 링크가 걸려있는 본래 파일 확인

```
# unshare --fork --pid --mount-proc readlink /proc/self
1
$ unshare --map-root-user --user sh -c whoami
root
# touch /root/uts-ns
# nsenter --uts=/root/uts-ns hostname
FOO
# umount /root/uts-ns/
```

UNSHARE(mnt)

컨테이너 생성시, OverlayFS를 다음과 같은 형태로 만든다.

```
# mkdir /root/mnt-ns/
# mount --bind /root/mnt-ns/ /root/mnt-ns/
# mount --make-private /root/mnt-ns
# touch /root/mnt-ns/mnt
# unshare --mount=/root/mnt-ns/mnt
```

UNSHARE(net)

컨테이너에서 사용하는 네트워크는 아래와 같이 생성한다. 네임스페이스 및

```
# touch /root/net-ns
# unshare --uts=/root/uts-ns hostname opensource
# unshare --net=/root/net-ns true
# ip link add veth0 type veth peer name veth1
# nsenter --net=/root/net-ns sleep 30 & pid=$!
# ip link set veth1 netns $pid
# ip -br link
```

UNSHARE(net)

```
# mount -t tmpfs --make-rshared tmpfs /run/netns
# mount -t tmpfs tmpfs /run
# ip netns add net-ns
# ip link add name veth0 type veth peer netns net-ns name veth1
# ip -br link
# ip -n net-ns -br link
```

UNSHARE(loopback device)

```
# unshare -n bash
# echo $$
# ip link
# touch /run/netns/new_namespace
# mount -o bind /proc/$$/ns/net /run/netns/new_namespace
# ip netns exec new_namespace bash
# ip link set lo up
# ip a l
```

UNSHARE(namespace isolate)

```
nsenter -t <PID> -n -S  100024 -G 100024
```

process ID

network

setuid

setgid

NS + SELinux + SECCOMP

PODMAN 설치하기

command

INSTALLATION PODMAN

포드만 설치는 RHEL기반의 클론 버전인 ROCKY LINUX에서 설치한다.

가상머신은 이미 oVirt시스템에 리눅스 설치 및 구성이 되어 있기 때문에 바로 설치만 진행하면 된다.

```
$ dnf install podman

$ podman images

$ podman pods ls

$ podman containers ls
```

COMMAND STRUCTURE

PS

```
$ podman container ls
$ podman ps
```

둘 다 동일한 컨테이너 목록을 출력. 권장은 'podman container ls'명령어 사용을 권장.

POD

```
$ podman pod ls
```

생성 및 동작중인 POD를 확인한다. 일반적으로 사용하는 POD는 쿠버네티스에서 사용하는 'pause'기반으로 POD를 구성한다.

POD

POD는 배포판 조금씩 다를 수 있다. 레드햇 계열은 RHLE 7이후 부터는 kata를 사용하고 있으며,
이전 버전들은 쿠버네티스에서 사용하는 pause기반으로 사용하고 있다.

https://katacontainers.io/

POD

```
$ podman pod start <ID> or <NAME>
POD ID        NAME              STATUS      CREATED        INFRA ID
# OF CONTAINERS
5de2fc474c1c  gallant_mahavira  Running     2 minutes ago  e223bc8ef0b7
1
```

catatonit

```
$ ps -ef | grep
POD ID        NAME              STATUS      CREATED        INFRA ID      # OF
CONTAINERS
5de2fc474c1c  gallant_mahavira  Running     2 minutes ago  e223bc8ef0b7  1
$ podman save <POD_CONTAINER_IMAGE> -o pod.tar
$ mkdir kata
$ tar xf pod.tar -C kata/
```

문제가 없으면 Pod는 kata기반으로 동작한다. RHLE 8/9 기반의 배포판은 kata기반으로 동작한다.

POD VERIFY

```
$ podman image save <ID> -o pause.tar
Copying blob d3462f6f15cc done
Copying config c7660493b5 done
Writing manifest to image destination
Storing signatures
[root@container ~]# ls
anaconda-ks.cfg  pause.tar
[root@container ~]# tar xf pause.tar -C /tmp/
```

RUN

```
$ podman run -ti --rm centos:stream9-minimal bash
```

런타임 밑으로 컨테이너를 실행한다.

-t: pseudo 장치로 컨테이너 tty를 구성 및 연결한다.

-i: 대화형 모드.

--rm: 컨테이너가 중지가 되면, 컨테이너를 제거한다.

bash: 맨 끝에 명령어는 컨테이너에서 실행하는 명령어를 적는다.

RUN

이미지가 없으면 먼저 이미지를 내려받기, 그리고 컨테이너를 runc나 혹은 kata기반으로 실행한다.

컨테이너 안에서 모든 명령어는 아니지만, 이미지에 미리 구성된 명령어 몇 가지를 실행 할 수 있다.

```
# podman run -it --rm centos:stream9-minimal bash
[root@9bab61b80300 /]# ip link
bash: ip: command not found
[root@9bab61b80300 /]# ls /bin | wc -l
185
```

RUN

```
[root@9bab61b80300 /]# cd /dev/
[root@9bab61b80300 dev]# ls
console  core  fd  full  mqueue  null  ptmx  pts  random  shm
stderr  stdin  stdout  tty  urandom  zero
[root@9bab61b80300 dev]#
```

RUN(PORT)

```
$ podman run -d -p 80:8080 --rm --name centos-httpd
centos7/httpd-24-centos7
```

-p: 컨테이너에서 사용하는 포트를 호스트로 바인딩한다. 왼쪽은 '컨테이너 포트', 오른쪽은 '호스트 포트'를 명시한다.

RUN(PORT)

```
$ podman run -d -p 8080:80 --rm --name centos-httpd
centos7/httpd-24-centos7
```

-p: 컨테이너에서 사용하는 포트를 호스트로 바인딩한다. 왼쪽은 '호스트 포트 ', 오른쪽은 '
컨테이너 포트 '를 명시한다.

RUN(PORT)

```
# podman port -l or port <CONTAINER ID/NAME>
80/tcp -> 0.0.0.0:8080
```

호스트 포트

컨테이너 포트

STOP

컨테이너를 중지하기 위해서 'stop'명령어를 실행한다.

```
$ podman stop <CONTAINER ID> <CONTAINER NAME>
$ podman container stop
```

중지가 잘 되었는지 'ps'명령어를 통해서 확인한다.

```
$ podman ps
$ podman container ps
```

STOP

실행중인 모든 컨테이너를 중지하기 위해서는 '--all'명령어를 통해서 중지가 가능하다.

```
$ podman container stop --all
```

올바르게 컨테이너가 중지되지 않으면 직접적으로 중지 신호를 명시 할 수 있다.

```
$ podman container stop -t <SIGKILL NUMBER> <CONTAINER ID OR
NAME>
```

START

중지된 컨테이너를 다시 실행하기 위해서 위의 명령어로 실행한다.

```
$ podman container start <ID> <CONTAINER NAME>
```

--all: 중지된 모든 컨테이너를 실행한다.

--attach: 터미널에 실행중인 컨테이너 표준출력을 연결한다.

-i(interactive): 대화형 형태로 표준입력을 연결한다.

INSPECT

- 동작중인 pod/container의 정보를 확인한다.
- 정보는 JSON형태로 제공한다.
- 이미지도 역시 inspect명령어로 확인이 가능하다.

컨테이너 런타임별로 동작 방법이 조금 다를 수 있으니, 주의 할 것.

```
$ podman image inspect <ID> <NAME>
         container
         pod
```

INSPECT

자주 사용하는 옵션은 다음과 같다.

--latest(-l): 마지막에 실행된 컨테이너의 정보를 확인한다. 이름 및 컨테이너 아이디 명시가 필요 없다.

--format: JSON에서 특정 데이터 딕셔너리를 선택하여 화면에 출력한다.

--size: 컨테이너가 사용하는 공간 크기를 같이 검사한다. image, pod에서는 이 옵션은 동작하지 않음.

INSPECT

JSON에서 특정 딕 및 리스트에서 값을 찾아서 화면에 출력한다.

```
$ podman container inspect centos-httpd --format '{{ .Config.Cmd }}'
```

[/usr/bin/run-httpd]

```
# podman container inspect centos-httpd --format '{{
.Config.StopSignal }}'
```

15

```
# podman container inspect centos-httpd --format '{{
.NetworkSettings.Ports }}'
```

map[8080/tcp:[{ 80}] 8443/tcp:[]]

PS

컨테이너화 되어서 동작하는 프로세스 목록을 확인한다.

```
# podman container ps
CONTAINER ID  IMAGE                                          COMMAND
CREATED        STATUS             PORTS                 NAMES
a98f2a616cfa  quay.io/xinick/containerlab/httpd:latest  /bin/sh -
c httpd ...  2 hours ago     Up 2 hours ago     0.0.0.0:8080-
>80/tcp  gallant_mahavira
```

PS

'ps' 명령어는 **'process'**의 약자이며, 컨테이너에서 동작중인 프로세스 목록을 확인한다.

--size: 현재 사용중인 컨테이너의 크기를 확인한다. 다만, 시간이 좀 걸린다.

--sort: 정렬 기준의 필드를 선택합니다.

--noheading: 맨 위의 제목 필드를 출력하지 않는다.

--latest(-l): 마지막에 생성된 컨테이너를 출력한다.

--watch(-w): 실시간으로 프로세스 목록을 명시된 초만큼 갱신한다.

RM

컨테이너를 제거한다. 다만, 컨테이너 제거 시, 컨테이너에서 사용하는 프로세스는 반드시 중지가 되어 있어야 한다.

```
# podman container rm centos-httpd
          pod
```

--all: 모든 컨테이너를 제거한다.

--force: 컨테이너를 강제로 제거한다. 다만, 사용 중이면, 컨테이너가 중지가 될 때 제거가 된다.

RM

런타임 엔진에서 관리하는 이미지를 제거한다.

```
# podman rmi <IMAGE NAME OR ID>
```

--all: 모든 이미지를 한번에 제거한다. 다만, 사용중인 경우 제거가 되지 않을 수 있으며, 사용 중에 강제로 제거하면 컨테이너가 올바르지 않는 동작을 한다.

EXEC

컨테이너 내부에 명령어 실행. 컨테이너에서 생성된 파일이나 혹은 바인딩된 디렉터리를 확인 시 사용한다.

```
# podman exec -i centos-httpd bash -c 'cat >
/var/www/html/index.html' << _EOF
Hello World
_EOF
```

EXEC

대화형 모드는 반드시 '-i', '-t'옵션을 통해서 접근이 가능하다.

```
# podman exec centos-httpd cat /var/www/html/index.html
# curl localhost:8080
```

RUN(EXEC)

컨테이너 실행자마다 다르지만, 컨테이너 실행자는 'tty', 'pts'를 지원하지 않는다.

```
# podman exec -it <POD_ID> -- /bin/bash
```

COMMIT

현재 사용중인 컨테이너를 이미지로 저장한다.

```
# podman stop centos-httpd
# podman commit centos-httpd commit-httpd
```

Getting image source signatures

Copying blob 53498d66ad83 skipped: already exists

Copying blob e41246bc487f skipped: already exists

Copying blob d320f5da4f70 skipped: already exists

COMMIT

커밋된 이미지를 다시 컨테이너로 실행한다.

```
$ podman run -d --name my-httpd -p8080:80 commit-httpd
```

CONTAINER IMAGE

Images are layerd on top of each other inheriting files from the lower layer images. Adding,removing and replacing files on lower level.

Lowest layer is called base image. Usually container libraries and package management tools to help create new layers.

PODMAN CONTAINER

컨테이너 로그를 확인한다. 컨테이너가 시작하면 표준 출력/오류를 런타임으로 전달한다.

```
# podman container logs centos-httpd
=> sourcing 10-set-mpm.sh ...
=> sourcing 20-copy-config.sh ...
=> sourcing 40-ssl-certs.sh ...
---> Generating SSL key pair for httpd...
AH00558: httpd: Could not reliably determine the server's fully qualified domain
name, using 10.88.0.19. Set the 'ServerName' directive globally to suppress this
message
[Sun Feb 05 06:47:31.399659 2023] [ssl:warn] [pid 1] AH01909: 10.88.0.19:8443:0
server certificate does NOT include an ID which matches the server name
```

PODMAN CONTAINER

컨테이너에 마운트 혹은 바인딩(mount --bind)이 된 자원을 확인한다.

```
$ podman container mount
77da2c1e24e8
/var/lib/containers/storage/overlay/1b003f63da8767dce7dacdcc43104
5ef46264a003047efeb9c032061e1e7506c/merged
a98f2a616cfa
/var/lib/containers/storage/overlay/9e781c2758890d0d7e9328cd6c696
74cec7a1c7840ed06e379d847637507abc1/merged
```

PODMAN CONTAINER

동작중인 컨테이너를 일시적으로 중지한다.

```
$ podman container pause centos-httpd
77da2c1e24e84283547e6b7defcdf80231b2833d28c18fa19bf5a0b222c88e85
```

```
$ podman container ls
CONTAINER ID  IMAGE                                          COMMAND
CREATED      STATUS          PORTS                   NAMES
a98f2a616cfa  quay.io/xinick/containerlab/httpd:latest  /bin/sh -c httpd ...
3 hours ago  Up 3 hours ago  0.0.0.0:8080->80/tcp  gallant_mahavira
```

PODMAN CONTAINER

일시 중지된 컨테이너를 다시 시작 시 사용한다.

```
$ podman container unpause
```

PODMAN CONTAINER

사용하지 않는 컨테이너를 모두 제거한다.

```
$ podman container prune
```

WARNING! This will remove all non running containers.

Are you sure you want to continue? [y/N]

PODMAN CONTAINER

컨테이너 이름을 변경한다.

```
$ podman container rename centos-httpd centos-apache
$ podman container ps
```

CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES

842293d27f7d quay.io/xinick/containerlab/httpd:latest /bin/sh -c httpd ... 29 seconds ago Up 29 seconds ago 0.0.0.0:8080->80/tcp centos-apache

PODMAN CONTAINER

동작중인 컨테이너를 중지 후 다시 시작한다.

```
$ podman container restart centos-apache
842293d27f7d4bfe62c433fb50bb1a477944335cd65a8d94ee97742be46b1b2b
```

PODMAN CONTAINER

현재 상태를 체크포인트로 저장 합니다.

```
$ podman container checkpoint --keep centos-httpd
1dcb10cfe96a65ffe63f2a8fba97d93ad7a1133b09372e5f4
d347c4a5cc737bf
```

PODMAN CONTAINER

특정 시점으로 컨테이너 상태를 변경합니다.

이미지나 혹은 동작중인 컨테이너에 적용이 가능합니다. 이 명령어는 앞서 사용한 checkpoint와 관련이 있습니다.

```
$ podman container restore centos-httpd
```

1dcb10cfe96a65ffe63f2a8fba97d93ad7a1133b09372e5f4d347c4a5cc737bf

PODMAN CONTAINER

이하 **'podman rm'**와 동일.

```
$ podman container rm
```

PODMAN CONTAINER

이하 'podman run'와 동일.

```
$ podman container run
```

PODMAN CONTAINER

```
$ podman container runlabel run
                   runlabel install
```

PODMAN CONTAINER

이하 'podman start'와 동일.

```
$ podman container start
```

PODMAN CONTAINER

컨테이너 상태를 확인 합니다. 이 정보는 systemd에서 cgroup으로 모니터링한다. 매 5초마다 정보를 갱신한다.

```
$ podman container stats
```

```
ID            NAME          CPU %       MEM USAGE / LIMIT  MEM %       NET IO          BLOCK IO      PIDS        CPU TIME    AVG CPU %
1dcb10cfe96a  centos-httpd  0.04%       32.48MB / 2.728GB  1.19%       796B / 1.836kB  32.77kB / 0B  213         179.744ms   0.09%
```

PODMAN CONTAINER

이하 'podman stop'과 동일.

```
$ podman container stop
```

PODMAN CONTAINER

네임스페이스에서 동작하는 프로세스의 CPU 사용률 및 동작 시간을 확인한다.

```
[root@container ~]# podman container top centos-httpd
USER        PID         PPID        %CPU        ELAPSED         TTY         TIME        COMMAND
root        1           0           0.000       7m2.357571898s  ?           0s          httpd -DFOREGROUND
apache      2           1           0.000       7m2.357666755s  ?           0s          httpd -DFOREGROUND
apache      3           1           0.000       7m2.357688937s  ?           0s          httpd -DFOREGROUND
apache      4           1           0.000       7m2.35770685s   ?           0s          httpd -DFOREGROUND
apache      5           1           0.000       7m2.357724764s  ?           0s          httpd -DFOREGROUND
```

```
$ podman container top <CONTAINER_ID> <NAME>
```

PODMAN CONTAINER

바인딩된 루트 파일시스템(root filesystem, /var/lib/containers/storage)디렉터리를 컨테이너에서 해제한다. 컨테이너 재시작은 필요 없으며, 네임스페이스 공간을 통해서 연결이 해제된다.

```
$ podman container unmount
$ podman inspect centos-httpd-volume --format '{{
.HostConfig.Binds }}'
[htdocs:/var/www/html/:rw,rprivate,nosuid,nodev,rbind]
$ podman container unmount centos-httpd-volume
```

PODMAN CONTAINER

컨테이너의 실행 상태를 코트로 화면에 출력한다. 시스템에서 사용하는 시스템 리턴 코드와 동일하게 0,1 그리고 125와 같은 숫자로 표현한다.

```
$ podman container wait
$ podman container wait --condition=running centos-httpd
-1
```

CONTAINER IMAGE

podman

IMAGE

```
$ podman image tree quay.io/xinick/containerlab/httpd
Image ID: 3a6f5259bcf9
Tags:     [quay.io/xinick/containerlab/httpd:latest]
Size:     220.2MB
Image Layers
├── ID: 2833ead9b90c Size: 160.1MB Top Layer of:
[quay.io/centos/centos:stream9]
└── ID: 6215263c40f2 Size: 60.06MB Top Layer of:
[quay.io/xinick/containerlab/httpd:latest]
```

IMAGE

```
$ podman image diff localhost/commit-httpd
quay.io/xinick/containerlab/httpd
C /bin
C /boot
A /boot/grub
A /boot/grub/splash.xpm.gz
C /opt
A /opt/rh
```

IMAGE LIST

```
$ podman images
REPOSITORY                          TAG             IMAGE ID      CREATED      SIZE
<none>                              <none>          3b7f1f5086e4  2 hours ago  220
MB
localhost/commit-httpd              latest          7fdd4880496d  5 hours ago  357
MB
quay.io/xinick/containerlab/httpd   latest          3a6f5259bcf9  8 hours ago  220
MB
quay.io/centos7/httpd-24-centos7    latest          33ab4360c736  4 days ago   356
MB
```

IMAGE

```
$ podman image inspect localhost/commit-httpd
[
     {
          "Id":
"7fdd4880496d7ed486ae5acd6a3f21af651ee11726d8a52744d3acc67897e074",
          "Digest":
"sha256:4ae5952af80c478ccc332fa0818a21176648a5eb124df0d490018a90c8450813",
          "RepoTags": [
               "localhost/commit-httpd:latest"
          ],
```

CONTAINER SHPPING

데모 및 시연

VIRTUAL VS CONTAINER

SERVICE ARCHITECTURE

ARCHITECTURE

- 포드만은 기존 모놀릭 및 마이크로 서비스 둘 다 구성이 가능하다.
- 하지만, 컨테이너는 기본적으로 stateless구조를 가져가기 때문에 기존 3Tiers는 어려운 부분이 있다.
- 기존에 사용하였던 3 Tiers 기반의 소프트웨어 구성은 컨테이너 환경에 전환이 완벽하지 않을 수 있다.

REGISTRY CONFIG

```
# vi /etc/containers/registries.conf
unqualified-search-registries = ["registry.access.redhat.com",
"registry.redhat.io", "docker.io", "quay.io"]
short-name-mode = "enforcing"
```

docker.io, registry.*.redhat.com 경우에는 계정이 없는 경우 다운로드에 제한이 있음.

런타임에서 사용하는 이미지는 전부 위의 설정의 주소에서 이미지를 내려받기 한다.

IMAGE CONFIG

```
# podman inspect centos-image
```

```
"GraphDriver": {
"Name": "overlay",

               "Data": {

                    "LowerDir":
"/var/lib/containers/storage/overlay/615c94009c7533a92dbf546ab10ba564a4e16a213400f3d351039d5686cc52c4/diff",

                    "MergedDir":
"/var/lib/containers/storage/overlay/f5502af5ecbc2bf3d0a89964ba099b8fb53232681de2d2d941082272daa37ff3/merged",

                    "UpperDir":
"/var/lib/containers/storage/overlay/f5502af5ecbc2bf3d0a89964ba099b8fb53232681de2d2d941082272daa37ff3/diff",

                    "WorkDir":
"/var/lib/containers/storage/overlay/f5502af5ecbc2bf3d0a89964ba099b8fb53232681de2d2d941082272daa37ff3/work"
}

          },
```

IMAGE CONFIG

LowerDir

맨 밑에 있는 이미지 레이어. 이미지 빌드 시 맨 처음에 구성한 베이지 레이어 및 기타 설치 및 복사한 파일들이나 혹은 패키지들로 구성 되어있다.

UpperDIR

상위 이미지 레이어, 컨테이너가 동작하면서 생성한 파일 및 디렉터리가 여기에 생성된다.

MergedDir

변경된 파일이나 혹은 추가된 파일은 MergedDir를 통해서 이미지 레이어에 추가 혹은 합쳐진다.

WorkDir

이 디렉터리는 일반적으로 비어 있다. OverlayFS 커널 모듈에서 사용한다.

OVERLAYER

Docker constructs
Container mount
Container layer
Image layer
file1
file2
file3
file4
file2
file4
file1
file2
file3
"merged"
"upperdir"
"lowerdir"
OverlayFS constructs

OVERLAYER

```
# mkdir /root/lower /overlay
# touch /root/lower/base_image.txt
# mount -oro,bind /root/lower /root/lower
# mount -obind /rootS/upper/ /overlay
# mkdir /overlay/{upper,work}
# mount overlay -t overlay \
-o lowerdir=/root/lower,\
upperdir=/overlay/upper,workdir=/overlay/work \
/media
```

IMAGE FORMAT

이미지 포맷 확인 방법은 두 가지가 있다.

- `podman inspect docker://quay.io/centos/centos:stream9-minimal`
- `skopeo inspect docker://quay.io/centos/centos:stream9-minimal`

여기서는 podman명령어로 이미지 정보를 확인한다.

RUN TEST CONTAINER

```
$ podman run -d --name centos-image \
quay.io/centos/centos:stream9-minimal \
/bin/sleep 10000
```

CONTAINER IMAGE DIRECTORY

컨테이너 생성 및 이미지 동작 시 이미지는 다음과 같은 위치에서 생성이 된다.

`/var/lib/containers/storage/overlay`

이 위치는 OCI IMAGE 사양에 정해진 위치. 역할 및 규칙에 따라서 파일이 생성이 된다.

CONTAINER IMAGE DIRECTORY

overlay

컨테이너가 동작하면(Pod, Container) 이 위치에 동작 시 필요한 컨테이너 디렉터리가 생성이 된다. 컨테이너에서 사용하는 읽기/쓰기가 가능한 디렉터리는 backingFsBlockDev를 통해서 연결이 된다.

연결이 되는 정보는 `'l'`이라는 디렉터리에 링크로 임시 레이어가 구성이 되어 있다.

CONTAINER IMAGE DIRECTORY

overlay-containers

컨테이너가 실행이 되면서 발생하는 디렉터리 및 파일을 해당 위치에 생성한다.

overlay-images

컨테이너에서 사용하는 이미지는 우리가 알고 있는 raw, qcow같은 이미지와 다르다. 이 위치에는 이미지 관련된 메타 정보가 저장이 되어 있다.

overlay-layers

이미지에서 사용하는 레이어 정보를 저장하고 있다.

IMAGE INSPECT

모든 컨테이너 정보 및 이미지 정보는 OCI(Open Container Initiative)사양으로 제작이 되어있다. 모든 정보는 JSON 형식 및 파일로 정보를 가지고 있다.

OCI사양은 기본적으로 도커 이미지 기반으로 되어있기 때문에 대다수 현재 컨테이너 이미지에 전부 적용이 가능하다.

DOCKER VS PODMAN

기능	PODMAN	DOCKER	
OCI 및 DOCKER 이미지 지원	지원	지원	
OCI 컨테이너 엔진 지원	지원	지원	
간단한 명령어 지원	지원	지원	
시스템 블록 통합(systemd)	지원	미지원	docker는 자체적으로 namespace, cgroup관리.
Fork/Exec 지원	지원	지원	
클라이언트 서버(API)	지원	지원	도커는 EE버전으로 지원하며 CE에서는 지원하지 않음.
docker-compose	지원	지원	

DOCKER VS PODMAN

기능	PODMAN	DOCKER	
레지스트리 수정	지원	미지원	podman은 보안, 저장소 및 네트워크 수정 가능
기본구성 설정 변경	지원	지원	
맥 지원	지원	지원	
윈도우 지원	지원	지원	WSL2 기반으로 지원
리눅스 지원	지원	지원	
업데이트 시 중지 필요	해당사항 없음	중지 필요	

RING STRUCTURE

RING STRUCTURE

check to rootless

rootless

UID/GID

/proc/self/loginuid

사용자 로그인 시 사용하는 uid 확인

/proc/self/gid_map

사용자 로그인 시 사용하는 gid 및 맵핑 된 namespace gid

**일반 사용자에서 반드시 컨테이너 생성 작업을 해야 됨. 루트에서 작업하는 경우 차이점 확인이 매우 어려움.

*** su, sudo명령어로 전환은 금지. 반드시, ssh명령어나 혹은 콘솔로 직접 접근해야 한다.

ROOTLESS

```
$ podman run -it --name hacker --privileged -v /:/host ubi8
chroot /host
```

/etc/group

컨테이너 런타임이 사용하는 사용자 그룹 정보. 사용하는 런타임마다 다르지만, 구형 런타임은 시스템 그룹정보를 사용한다.

ROOTLESS

/etc/subgid

OCI사양의 런타임 구성파일. 사용자 gid를 subgid에 명시된 값으로 다시 맵핑한다. 이전 도커에서 사용하는 /etc/group 맵핑과 비슷하지만 별도 파일로 선언한다.

/etc/subuid

OCI사양의 런타임 구성파일. 사용자 uid를 subuid에 명시된 값으로 다시 맵핑한다. 이전 도커에서 사용하는 /etc/group 맵핑과 비슷하지만 별도 파일로 선언한다.

/proc/self/loginuid

```
podman run quay.io/centos/centos:stream9-minimal cat
/proc/self/loginuid
```

ROOTLESS

"/proc/self/loginuid" 위의 정보 확인하기

```
$ podman run --rm quay.io/centos/centos:stream9-minimal cat
/proc/self/loginuid
```

ROOTFUL

```
$ sudo podman run --rm --privileged -v /:/host
centos/centos:stream9-minimal touch /host/etc/shadow
```

```
$ grep auid /var/log/audit/audit.log | tail -f
```

docker and runtimes

아마존 기초

DOCKER VS PODMAN

CONTAINER RUNTIMES(OCI/CRI)

CONTAINER INTERACTIVE(docker/containerd)

CRI STANDARD (DOCKER/CONTAINERD)

DOCKER ENGINE MODELS

CONTAINER+KUBELET

OCI RUNTIMES

OCI RUNTIMES

KUBELET RUNTIME

컨테이너 기초

KUBELT + CRI STANDARD

KUBELET + CRI STANDARD

KUBELET TOP/DOWN LAYERS

POD and CONMON

runtime

CONTAINER MONITOR

INFRA CONTAINER

RUNTIME

podman

RUNTIME

RUNC
conmon
CONTAINER HOST
run/create
APP
CONTAINER
POD
CRIO runtime
kernel
cgroup
seccomp
namespace

RUNTIME

PODMAN

podman

PODMAN

포드만은 docker의 대안으로 나온 컨테이너 런타임.

컨테이너 런타임은 데몬리스(daemonless)의 컨테이너 관리자이다.

자체적으로 API서비스를 가지고 있으며, 필요가 없는 경우 컨테이너 데몬을 실행할 필요가 없다.

PODMAN

포드만은 도커와 호환성을 제공한다.

io.podman.service, podman.service 데몬은 API를 제공.

- docker-compose
- docker-build
- volume
- network

PODMAN

- 포드만은 쿠버네티스 런타임으로 사용은 불가능함.
- 독립적인 구성만 가능하며, API기반으로 'DOCKER-EE'와 같은 기능을 제공함.

PODMAN

What is Podman? Podman is a daemonless container engine for developing, managing, and running OCI Containers on your Linux System. Containers can either be run as root or in rootless mode. Simply put: alias docker=podman

PODMAN

현재 사용하는 모든 컨테이너 시스템은 다음과 같은 표준을 따르고 있다. Podman(이하 앞으로 포드만)은 아래 사양을 따르고 있다.

- OCI, Open Container Initiative
- CNI, Container Network Interface
- CSI, Container Storage Interface
- CRI, Container Runtime Interface

DOCKER

- 도커는 DOCKER CE, EE 두 가지 버전으로 지원.
- 현재는 Mirantis라는 회사로 인수 및 개발중지.
- DOCKER-CE는 Containerd로 분리가 되었음.
- Containerd는 현재 CRI/OCI기본 컨테이너 런타임 및 이미지 표준.

DOCKER

```
     | <ovirt>            <kubernetes>
                                  |
-----------[middleware]------ | -
[USER]                            |
                        +--------+----+
        <QEMU>          |OCI Spec     |
        <PODMAN>        +-------------+
                        |<cri-o>      |
                        |<containerd> |
                        +--------+----+
-----------[systemd]--------- | -
                \             |
[KERNEL]         \      +-------+--+
        kvm.ko    \     |namespace |
          |        `---|cgroup    |
          |             |seccomp   |
      <hardware>        +----------+
```

DOCKER CE/EE

GUI
CLI
DTR
Scanning
Mirroring
Promotion
REST API
Classic API
Swarm API
Kube API
Router + Adapter
UCP Controller
CA
AuthN
AuthZ
DCT
Metrics
REST API
Classic API
Swarm API
Classic Swarm Mgr
Kube API
CNI
Kube Mgr
swarm scope network plugin
3rd party plugins
UCP Agent
etcd
Kubelet
REST API
Classic API
Swarm API
swarm-scheduler
secrets
inventory
raft
Docker EE Engine (managers)
Docker EE architecture
Kubernetes items shown in green
Highlights:
Exposes the full Kubernetes API
Backward compatible with EE Engine 17.06
Interop between pods, services, and containers
Docker daemon is the source of truth for resource usage for schedulers
Docker EE worker nodes
UCP Agent
Kubelet
REST API
Classic API
Swarm API
Docker EE Engine (workers)

DOCKER CE/EE

EE Engine
Certified Plugins Infra ISVs
Support SLA
FIPS
CE Engine
Docker CLI
dockerd
Docker API
Storage Mgmt
libnetwork
Docker Build
SwarmKit
Docker Content Trust
Image Mgmt
containerd
runc
Plugins Storage Networking
Docker Compose

CONTAINERD(FROM DOCKER)

CONTAINERD(FROM DOCKER)

NETWORK BPF

Enhanced BPF
also known as just "BPF"
User-Defined BPF Programs
SDN Configuration
DDoS Mitigation
Intrusion Detection
Container Security
Observability
...
Kernel
Runtime
Event Targets
verifier
BPF
BPF actions
sockets
kprobes
uprobes
tracepoints

TRACING TIMELINE

A Linux Tracing Timeline

- 1990's: Static tracers, prototype dynamic tracers
- 2000: LTT + DProbes (dynamic tracing; not integrated)
- 2004: kprobes (2.6.9)
- 2005: DTrace (not Linux), SystemTap (out-of-tree)
- 2008: ftrace (2.6.27)
- 2009: perf (2.6.31)
- 2009: tracepoints (2.6.32)
- 2010-2016: ftrace & perf_events enhancements
- 2014-2016: BPF patches

also: LTTng, ktap, sysdig, ...

BPF/eBPF/XDP

systemd

BPF

Berkeley Packet Filter(BPF)의 약자. BSD계열에서 사용하고 있는 기능을 리눅스로 이식하여 현재 사용하고 있음. BPF는 커널 3.15에서 도입이 되었으며, 개발은 1992년도에 되었음.

BPF

기존 BPF는 커널 수준에서만 필터링 하였다. 하지만, 가상머신 및 컨테이너 같은 시스템이 활성화 되면서, 커널에서 필터링을 할 수 없는 부분이 발생하였고, 이를 확장하기 시작하였다. 그래서 기존에 사용하던 BPF는 cBPF(Classic Berkeley Packet Filter)라고 부르기 시작하였다. 이는 32비트 영역만 지원한다.

이를 eBPF(Extended Berkeley Packet Filter)라는 이름으로 수정하였고, 사용자 영역 및 32비트 이상의 데이터도 접근 및 핸들링 할 수 있도록 하였다.

BPF/eBPF

BPF, eBPF는 다음과 같은 기능을 제공한다.

하드웨어 기반 가속기능(NIC)

CPU에서 제공하는 암호화 및 소수점 기능을 통한 가속화

XDP

BPF 및 eBPF는 기본적으로 BSD에 구성된 기능이기 때문에 BSD 커널에서 다루지 않는 부분은 사용하기 어렵다.

그래서 리눅스는 XDP(eXpress Data Path)프로젝트를 통해서 데이터 부분에 대한 필터링도 지원한다. XDP를 사용하는 경우 하드웨어가 BPF기능을 지원하지 않아도 XDP를 통해서 필터링이 가능하다.

BPFs

Packet Reflection Performance (1 core)
Socket Layer
TCP Stack
Kernel Space
1 Mpps
Connection Tracking
cBPF
eBPF
5 Mpps
Traffic Control
cBPF
eBPF
Driver Space
20 Mpps
XDP
(eXpress Data Path)
eBPF
Offload
74 Mpps
(line rate)
NIC
Other Networking
(significant complexity hidden here)
High Performance Networking Hooks

XDP

RX CPU
Other CPUs
Application
Control Application
Load/Configure BPF
Application
Sockets
XDP Packet Processor
Packet Steering
(RPS/RFS)
Sockets
TCP/IP Stack
Drop
Receive Local
Forward
TCP/IP Stack
GRO
Parsing/Processing BPF Program
Driver/Device

BPF/eBPF

Classical BPF Machine
cBPF Bytecode
Registers
0
31
Accumulator
X
Scratch Pad
0
31
15
Extended BPF Machine
eBPF Bytecode
Registers
Stack
Maps
General Purpose Registers
0
63
9
10
Stack Pointer
0
7
511
Key
Value
No SW limit to size

BPF LINUX VERSION

BPF Enhancements by Linux Version

- 3.18: bpf syscall
- 3.19: sockets
- 4.1: kprobes
- 4.4: bpf_perf_event_output
- 4.6: stack traces
- 4.7: tracepoints
- 4.9: profiling

eg, Ubuntu:

16.04

16.10

BPF LINUX VERSION

IPTABLES/NFTABLES/BPF

RUNC/OCI

systemd

OCI

OCI는 컨테이너를 관리하는 영역이다. 컨테이너 프로그램이 실행이 되면 모든 컨테이너는 OCI를 통해서 실행이 된다.

앞서 이야기 하였지만, 포드만이 컨테이너 라이프 사이클을 관리하며, 컨테이너 실행은 이래 프로그램들이 담당한다.

- runc
- crun
- kata
- gVisor

RUNC

OCI는 컨테이너를 관리하는 영역이다. 컨테이너 프로그램이 실행이 되면 모든 컨테이너는 OCI를 통해서 실행이 된다. 대표적인 관리 프로그램은 다음과 같다.

RUNC

RUNC는 Go-Lang기반으로 작성된 컨테이너 실행자. 현재 대다수 리눅스 배포판의 컨테이너는 RUNC기반으로 동작한다.

CRUN

crun

CRUN

RUNC와 동일한 기능을 제공하나, CRUN보다 가볍고 빠르게 실행한다.

하지만, 모든 상황에서 CRUN사용할 수 없기 때문에, 사용을 원하는 경우 OCI지원 사양을 확인 후 사용 결정을 하는게 좋다.

CRUN

- A fast and low-memory footprint OCI Container Runtime fully written in C.
- crun conforms to the OCI Container Runtime specifications (https://github.com/opencontainers/runtime-spec).

CRUN 성능

	crun	runc	%
100 /bin/true	0:01.69	0:3.34	-49.4%

Kata

Kata

Kata는 OCI 호환이 되는 컨테이너 실행자이다.

현재 Kata를 기본적으로 사용하는 컨테이너 런타임은 레드햇 오픈 시프트 CoreOS기반에서 사용하고 있다.

gVisor

gVisor

gVisor는 GO언어로 작성된 컨테이너 실행자이다. Kata처럼 OCI와 호환이 되며, 'runsc'를 통해서 커널과 애플리케이션의 영역을 격리한다.

gVisor

CRI and gVisor
Manage container images
CRI
gVisor
Expand file system
Init process
Container
Container
Container
gVisor's guest kernel
Host machine (with Host Linux kernel)

OCI APPLICATION LAYER

POD EMULATE

systemd

POD EMULATE #1

구성 조건

CREATE NAMESPACE

```
ip netns add redapp
ip netns add blueapp
ip netns
```

NAMESPACE WITH BRIDGE

```
# ip link set dev bridge-if1 up
# ip link
# ip link add veth-red1 type veth peer name veth-red1-br1
# ip link set veth-red1-br1 master bridge-if1
# ip link set veth-red1 netns redapp
# ip link set veth-blue1 netns blueapp
```

VIRTUAL DEVICE WITH NAMESPACE

```
# ip -n redapp addr add 192.168.1.11/24 dev veth-red1
# ip -n blueapp addr add 192.168.1.12/24 dev veth-blue1
# ip -n blueapp link set veth-blue1 up
# ip link set dev veth-blue1-br1 up
# ip link set dev veth-red1-br1 up
```

SET GATEWAY

```
# ip netns exec blueapp ip route add default via 192.168.1.10 dev
veth-blue1

# ip netns exec redapp ip route add default via 192.168.1.10 dev
veth-red1
```

NAT NODE1

```
# iptables -t nat -A POSTROUTING -s 192.168.1.0/24 -d
10.130.0.10/32 -j MASQUERADE
```

NODE1 TEST

```
# dnf install python3 -y
# ip netns exec redapp python3 -m http.server 8000
# ip netns exec redapp curl http://127.0.0.1:8000
# ip netns exec blueapp ping 192.168.1.11
# ip netns exec blueapp curl http://192.168.1.11:8000
```

NODE2 TEST

```
# ip netns exec blueapp ping 192.168.1.11
# ip netns exec blueapp curl http://192.168.1.11:8000
# ip netns exec redapp ping 192.168.1.11
# ip netns exec redapp curl http://192.168.1.11:8000
```

NODE1

```
# iptables -t nat -A PREROUTING -j DNAT -p tcp --dport 30001 -d
10.128.0.2 --to-destination 192.168.1.11:8000
```

VOLUME

systemd

VOLUME VS BINDING

도커 혹은 포드만 두 가지 형태로 스토리지를 제공한다.

- CLI에서 명령어로 호스트 자원을 바인딩하는 방법.
- 런타임에서 볼륨 생성 후, 컨테이너에 전달하는 방법.
  - /var/lib/containers/storage/volumes

POD
(infra container)

systemd

as .service under systemd

systemd

COMPABILITY K8S

systemd

SECURITY

systemd

BUILDAH

systemd

Dockerfile

Containerfile

SKOPEO

systemd

IMAGE SEARCH

IMAGE COPY

MIRRO

REGISTERY

systemd

REGISTRY

HABOR

QUAY

CONTAINER ORCHESTRATION

systemd

K8S

systemd

KUBERNETES DIAGRAM

KUBERNETES COMPONENT

KUBERNETES ABSTRACTION

HYPERVISOR VS RUNTIME

Traditional Deployment

Virtualized Deployment

Container Deployment

KUBERNETES FLOW

KUBERNETES CLUSTER MAP

KUBERNETES SERVICE ARCHITECTURE

KUBERNETES SERVICE ARCHITECTURE

OPENSHIFT

OPENSHIFT LAYERS

OPENSHIFT MIDDLEWARE LAYER

OPENSHIFT VERSION SHIFT

OPENSHIFT INSTALLATION

OPENSHIFT INSTALL TOOLS

OPENSHIFT UPDATE

OPENSHIFT COMPONENT

ROUTE AND API

OPENSHIFT SCALING FOR NODE

OPENSHIFT CI/CD

KUBERNETES(deployment+replicaset)

KUBERNETES(API SERVER)