

# Container부터 다시 살펴보는 Kubernetes Pod 동작 원리

전호준 (<https://hyojun.me>)

# What is Kubernetes Pod?

# What is Kubernetes Pod?

- 쿠버네티스에서 배포할 수 있는 최소 객체 단위
- 1개 이상의 컨테이너로 이루어진 그룹

# What is Kubernetes Pod?

- 쿠버네티스에서 배포할 수 있는 최소 객체 단위
  - 1개 이상의 컨테이너로 이루어진 그룹
- ➔ 컨테이너를 “잘” 알아야, Pod을 이해할 수 있습니다.

컨테이너는 “격리된 환경”에서 실행되는 “프로세스”

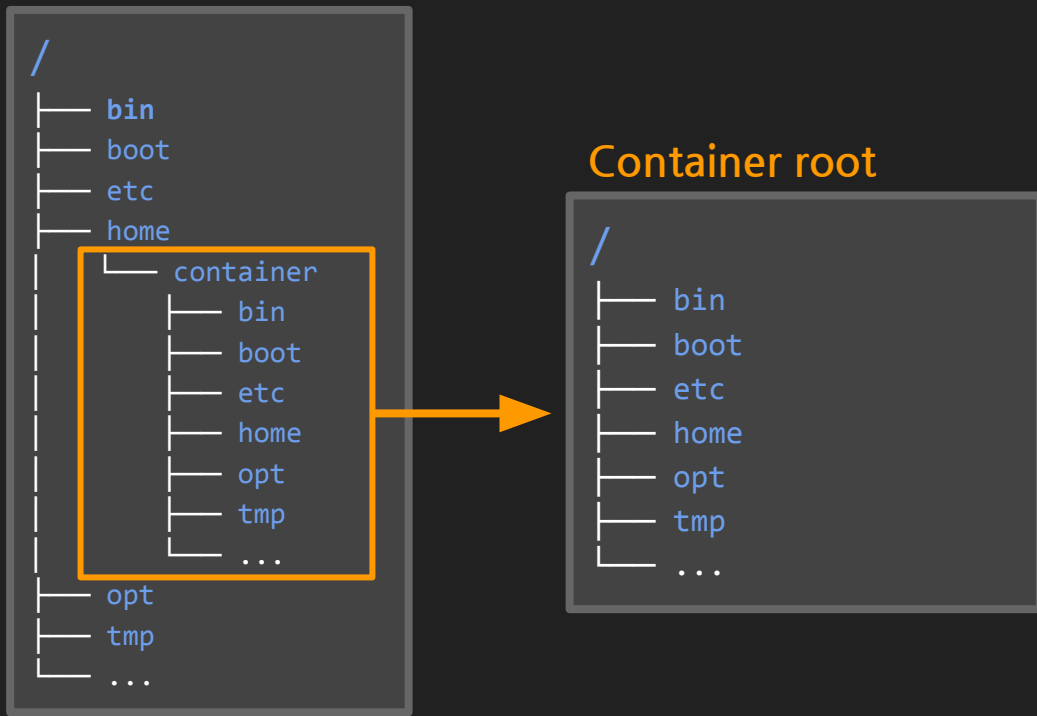
# 컨테이너가 격리된 환경을 구현하는 주요 원리

- 루트 디렉토리 격리(chroot)
- Linux namespaces
  - Mount (mnt)
  - Process ID (pid)
  - Network (net)
  - Interprocess Communication (ipc)
  - Unix Time-Sharing (uts)
  - User ID (user)
- Control group (cgroup)
- OverlayFS
- ... 등

→ 지금부터 알아보시다.

# 컨테이너가 격리된 환경을 구현하는 주요 원리

## (1) 루트 디렉토리 격리



# 컨테이너가 격리된 환경을 구현하는 주요 원리

## (1) 루트 디렉토리 격리

```
05:16ubuntu@ubuntu>container> pwd
/home/ubuntu/workspace/container-lecture/home/container
05:16ubuntu@ubuntu>container> tree -L 2 ./
./
├── bin
│   ├── bash
│   └── ls
├── lib
│   ├── libc.so.6
│   ├── libdl.so.2
│   ├── libpcr.so.3
│   ├── libpthread.so.0
│   ├── libselinux.so.1
│   ├── libtinfo.so.5
│   └── lib64
│       └── ld-linux-x86-64.so.2
3 directories, 9 files
```

bash / ls 명령어 바이너리

bash / ls 명령어 실행을 위한 의존성

```
05:16ubuntu@ubuntu>container> sudo chroot ./ /bin/bash
bash: warning: setlocale: LC_ALL: cannot change locale (en_US.UTF-8)
bash-4.4# ls
bin lib lib64
```

## chroot

입력된 경로(New root path)가 루트 디렉토리로 격리된 프로세스(command)를 실행하기 위한 명령어

```
$ chroot <NEWROOT> <COMMAND>
```



# 컨테이너가 격리된 환경을 구현하는 주요 원리

## (2) Linux Namespaces

### Linux Namespace

→ 프로세스 간 시스템 자원들을 격리하기 위한 Linux 커널의 기능

`$ lsns -p <pid>`

→ 실행 중인 프로세스의 namespace를 조회하는 명령어

```
06:26ubuntu@ubuntu>container> sudo lsns -p 95359
```

	NS	TYPE	NPROCS	PID	USER	COMMAND
4026531835		cgroup	150	1	root	/sbin/init maybe-ubiquity
4026531837		user	150	1	root	/sbin/init maybe-ubiquity
4026532493		mnt	1	95359	root	sleep 3600
4026532494		uts	1	95359	root	sleep 3600
4026532495		ipc	1	95359	root	sleep 3600
4026532496		pid	1	95359	root	sleep 3600
4026532498		net	1	95359	root	sleep 3600

호스트 PID=1의 namespace를  
그대로 사용

별도의 격리된 namespace  
사용

# 컨테이너가 격리된 환경을 구현하는 주요 원리

## (2) Linux Namespaces

### unshare

→ 특정 namespace 를 격리한 프로세스를 실행할 수 있는 명령어

```
# mount namespace 격리(-m)한 /bin/bash 프로세스 실행
$ unshare -m /bin/bash
# mount namespace(-m)와 ipc namespace를 격리(-i)한 /bin/bash 프로세스 실행
$ unshare -m -i /bin/bash
```

# 컨테이너가 격리된 환경을 구현하는 주요 원리

## (3) Mount(mnt) namespace

### Mount

Unix 계열 시스템에서 특정 파일 시스템에 접근하기 위해, 파일 시스템을 root 디렉토리(/)부터 시작하는 file tree의 특정 디렉토리에 연결하는 것

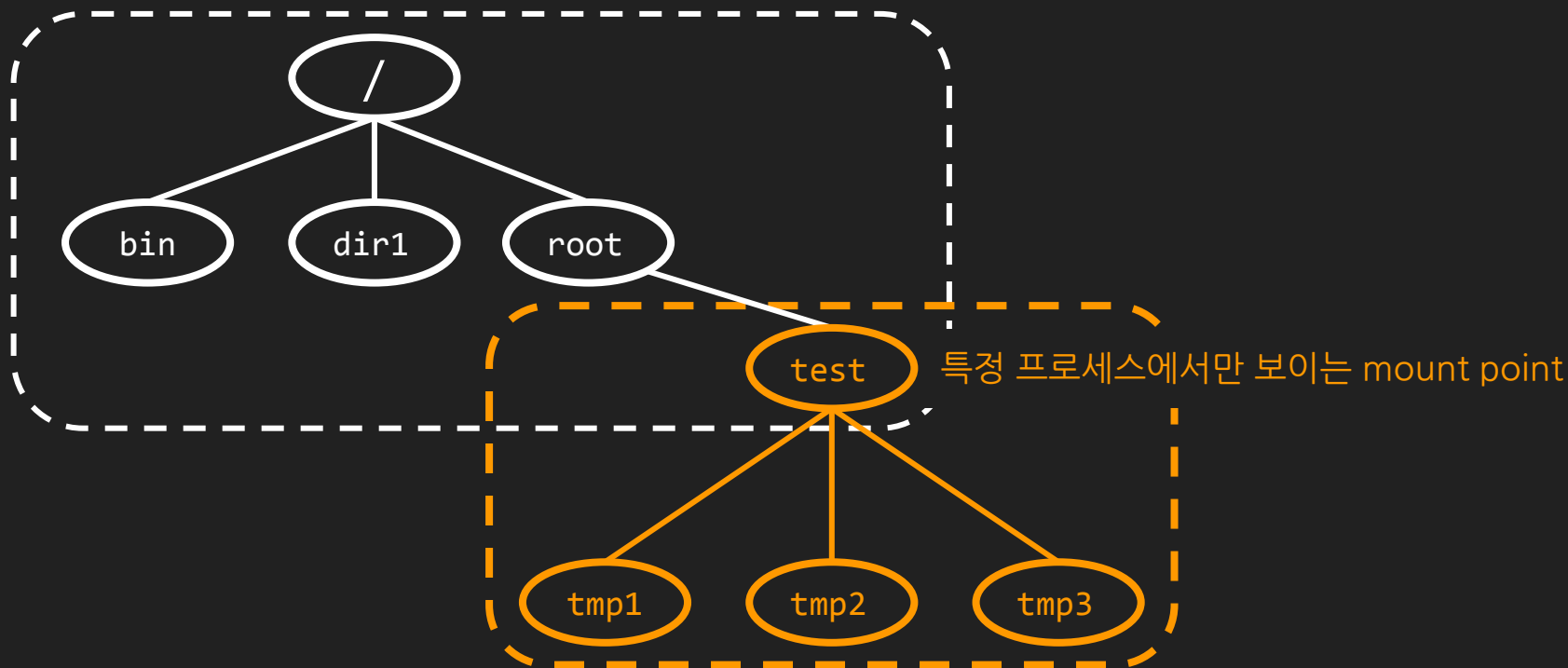
```
# mount -t <type> <device> <dir>
```

```
# 예) tmpfs(temporary file storage)를 “/root/test” 경로에  
마운트
```

```
$ mount -t tmpfs tmpfs /root/test
```

# 컨테이너가 격리된 환경을 구현하는 주요 원리

## (3) Mount(mnt) namespace



# 컨테이너가 격리된 환경을 구현하는 주요 원리

## (3) Mount(mnt) namespace

### Mount namespace

프로세스 간 서로 다른 mount point를 가질 수 있게 함

```
$ echo $$  
1111  
$ unshare -m /bin/bash  
$ echo $$  
2222  
$ mkdir -p test && mount -t tmpfs tmpfs ./test  
$ df -h | grep test  
tmpfs          2.0G      0  2.0G   0% /root/test  
$ exit  
$ df -h | grep test
```

현재 process ID 출력

격리된 Mount namespace(`-m` option)로 bash 실행

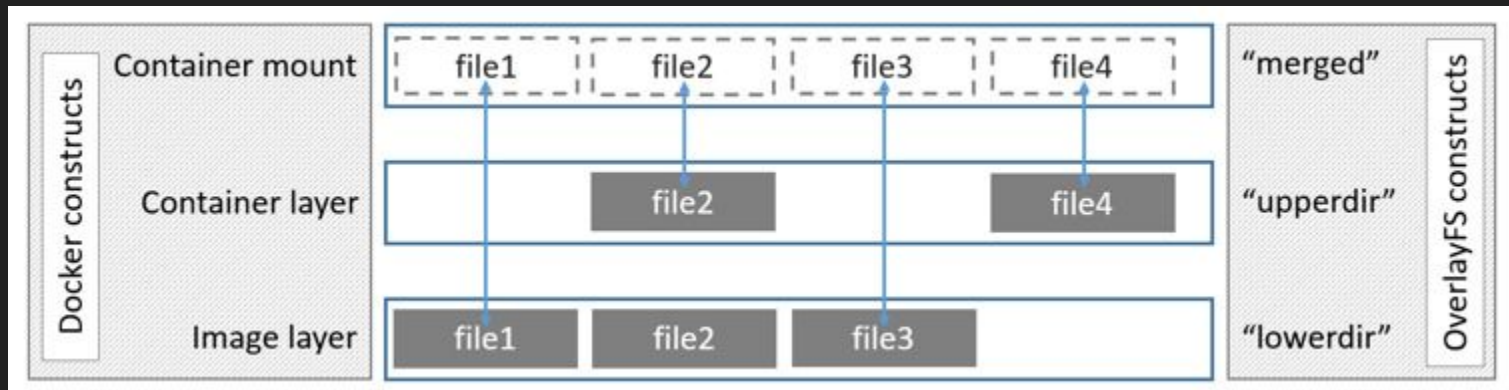
tmpfs(임시 파일 스토리지)를  
`/root/test`에 mount

bash 종료하고 파일시스템 확인  
결과) `/root/test`에 마운트한 파일 시스템이 보이지 않아야 함

# 컨테이너가 격리된 환경을 구현하는 주요 원리

## (3) Mount(mnt) namespace + OverlayFS

### OverlayFS storage driver

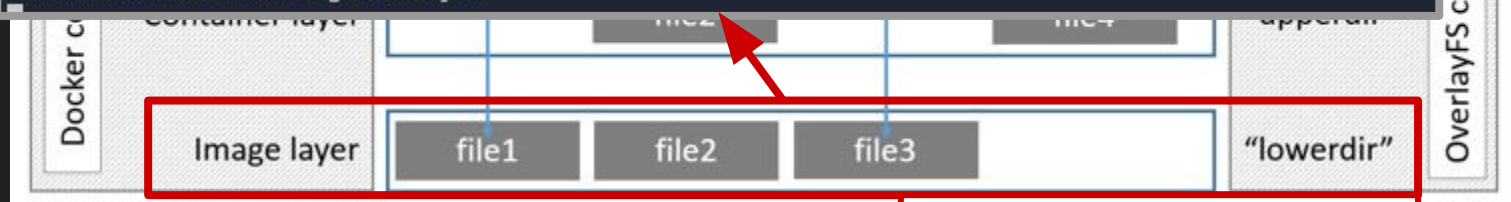


# 컨테이너가 격리된 환경을 구현하는 주요 원리

## (3) Mount(mnt) namespace + OverlayFS

### OverlayFS storage driver

```
00:16ubuntu@ubuntu>~> docker run --name ubuntu -d ubuntu:18.04 /bin/bash -c "sleep 3600"  
Unable to find image 'ubuntu:18.04' locally  
18.04: Pulling from library/ubuntu  
92dc2a97ff99: Pulling fs layer  
be13a9d27eb8: Pulling fs layer  
c8299583700a: Pulling fs layer
```

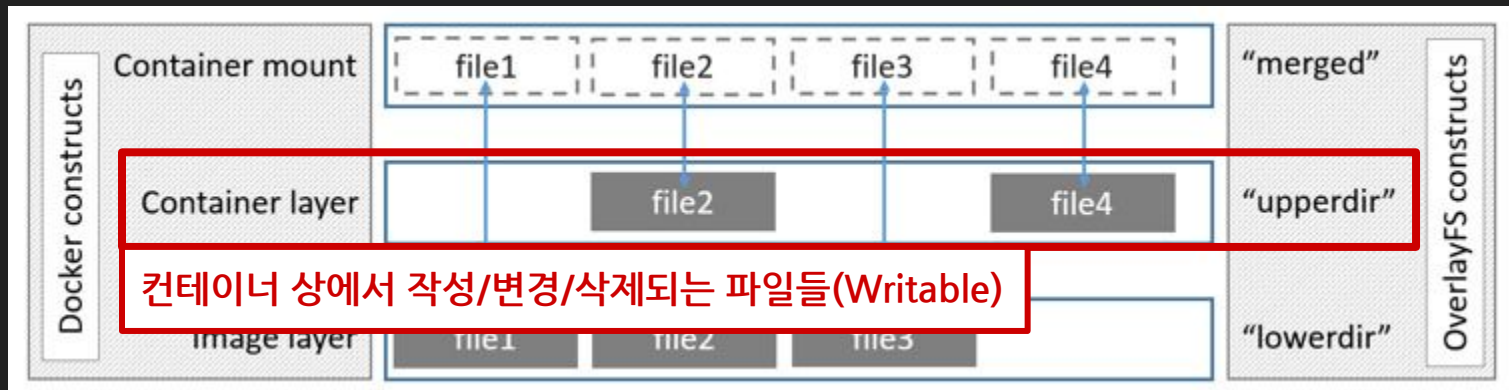


컨테이너 이미지(Read Only)

# 컨테이너가 격리된 환경을 구현하는 주요 원리

## (3) Mount(mnt) namespace + OverlayFS

### OverlayFS storage driver





# 컨테이너가 격리된 환경을 구현하는 주요 원리

## (3) Mount(mnt) namespace + OverlayFS

### OverlayFS storage driver

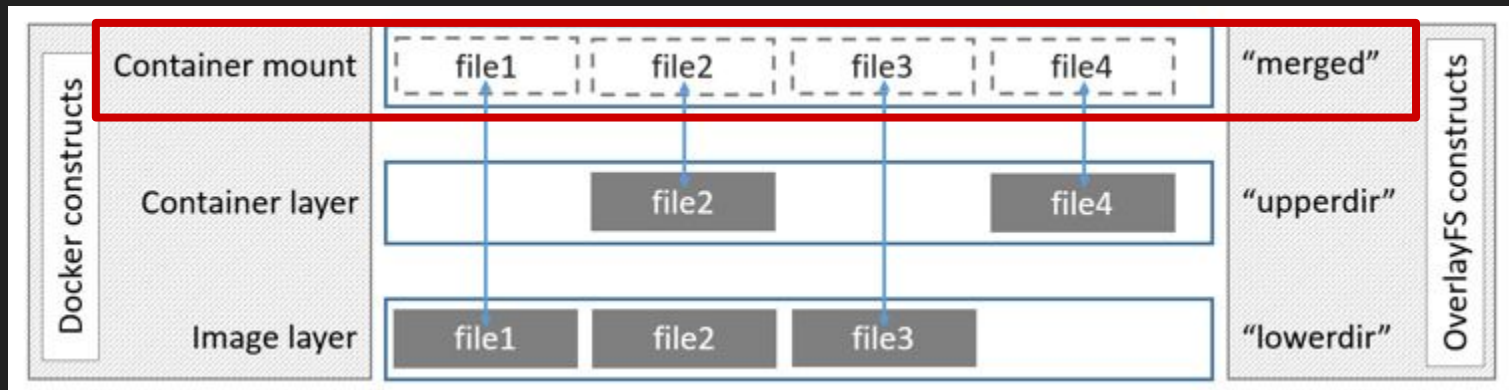


Image Layer + Container Layer = Merged = 최종적으로 OverlayFS에 반영

# 컨테이너가 격리된 환경을 구현하는 주요 원리

## (3) Mount(mnt) namespace + OverlayFS

### OverlayFS storage driver

```
$ docker inspect ubuntu | jq ".[].GraphDriver"
{
  "Data": {
    "LowerDir": "/var/lib/docker/overlay2/.../diff",
    "MergedDir": "/var/lib/docker/overlay2/.../merged",
    "UpperDir": "/var/lib/docker/overlay2/.../diff",
    "WorkDir": "/var/lib/docker/overlay2/.../work"
  },
  "Name": "overlay2"
}
```

# 컨테이너가 격리된 환경을 구현하는 주요 원리

## (3) Mount(mnt) namespace + OverlayFS

```
00:34xubuntu@ubuntu>~> docker exec -ti ubuntu /bin/bash
root@84c2fb9b121e:/# df -h
Filesystem      Size  Used Avail Use% Mounted on
overlay          98G   25G   69G   26% /
tmpfs            64M    0    64M    0% /dev
tmpfs            2.0G    0   2.0G    0% /sys/fs/cgroup
shm              64M    0    64M    0% /dev/shm
/dev/sda2        98G   25G   69G   26% /etc/hosts
```

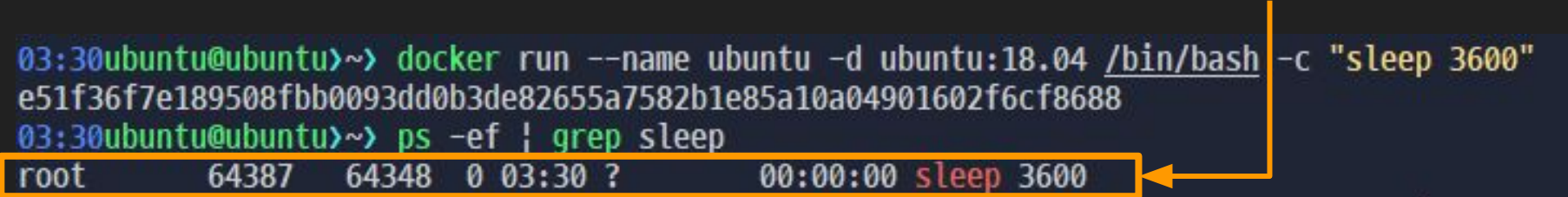
# 컨테이너가 격리된 환경을 구현하는 주요 원리

## (4) Process ID (pid) namespace

컨테이너는 “격리된 환경”에서 실행되는 “프로세스”

```
03:30ubuntu@ubuntu>~> docker run --name ubuntu -d ubuntu:18.04 /bin/bash -c "sleep 3600"
e51f36f7e189508fbb0093dd0b3de82655a7582b1e85a10a04901602f6cf8688
03:30ubuntu@ubuntu>~> ps -ef | grep sleep
```

root	64387	64348	0	03:30	?	00:00:00	sleep 3600
------	-------	-------	---	-------	---	----------	------------



컨테이너 안에서는 하나의 가상머신처럼 보이지만,  
컨테이너 밖(호스트)에서는 프로세스일 뿐

# 컨테이너가 격리된 환경을 구현하는 주요 원리

## (4) Process ID (pid) namespace

같은 프로세스에 대해,

호스트에서 확인한 PID와 컨테이너 안에서 확인한 PID가 다르다.

```
07:59ubuntu@ubuntu>~> docker run --name ubuntu -d ubuntu:18.04 /bin/bash -c "sleep 3600"  
d1d0b10115f927624075441703324fa99d756d833045fbb6108ddb7ba5fd5bf5
```

```
07:59ubuntu@ubuntu>~> ps -ef | grep sleep
```

```
root      115679  115645  0 07:59 ?        00:00:00 sleep 3600  
ubuntu    115734  115095  0 07:59 pts/1    00:00:00 grep --color=auto --exclude-dir=.b  
=.svn --exclude-dir=.idea --exclude-dir=.tox sleep
```

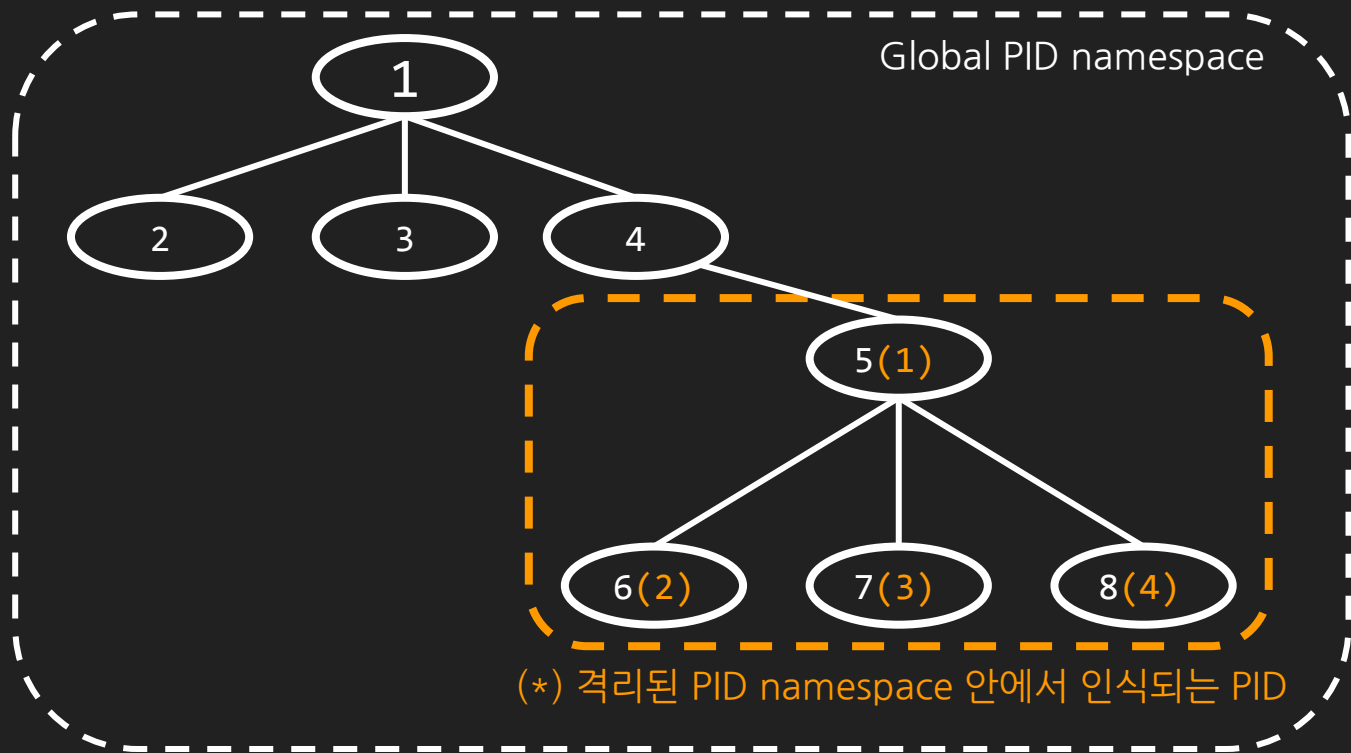
```
07:59ubuntu@ubuntu>~> docker exec ubuntu ps
```

```
  PID TTY          TIME CMD  
   1 ?            00:00:00 sleep  
   8 ?            00:00:00 ps
```

컨테이너 밖 PID=115679  
컨테이너 안 PID=1

# 컨테이너가 격리된 환경을 구현하는 주요 원리

## (4) Process ID (pid) namespace



# 컨테이너가 격리된 환경을 구현하는 주요 원리

## (4) Process ID (pid) namespace

PID namespace를 격리(`-p` option)한 bash 프로세스 실행

```
$ unshare -f -p /bin/bash
```

```
$ echo $$ # 현재 PID 출력
```

```
1
```

PID namespace가 격리된 컨테이너에서  
최초로 실행된 프로세스(entrypoint)는 항상 PID=1을 갖는다.

# 컨테이너가 격리된 환경을 구현하는 주요 원리

## (5) Inter-Process Communication (ipc) namespace

System V 기반의 프로세스 간 통신을 격리

- System V IPC
  - 공유 메모리(shm)
  - 세마포어
  - POSIX 메시지 큐(/proc/sys/fs/mqueue)
- IPC 객체들은 같은 IPC namespace에 존재하는 프로세스에만 표시



# 컨테이너가 격리된 환경을 구현하는 주요 원리

## (6) Network (net) namespace

네트워크 인터페이스, 라우팅, 방화벽 규칙들을 격리

```
# “test-ns” 이름의 network namespace 생성
```

```
$ ip netns add test-ns
```

```
$ ip netns list
```

```
test-ns
```

```
# Virtual ethernet interface pair 생성 (veth1, veth2)
```

```
# “veth1”은 test-ns에 생성, “veth2”는 PID 1의 network namespace에 생성
```

```
$ ip link add veth1 netns test-ns type veth peer name veth2 netns 1
```

# 컨테이너가 격리된 환경을 구현하는 주요 원리

## (6) Network (net) namespace

### 네트워크 인터페이스, 라우팅, 방화벽 규칙들을 격리

# 새로 생성된 “test-ns”에서 “ip link list” 명령어 실행 (네트워크 인터페이스 출력)

# test-ns에 할당한 veth1과 loop back 인터페이스만 존재

```
$ ip netns exec test-ns ip link list
```

```
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN mode DEFAULT group default qlen 1000  
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
```

```
8: veth1@if7: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000  
    link/ether 2a:aa:60:ee:27:d4 brd ff:ff:ff:ff:ff:ff link-netnsid 0
```

# 호스트의 기본 network namespace에서 “ip link list”를 실행

```
$ ip link list
```

```
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1000  
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
```

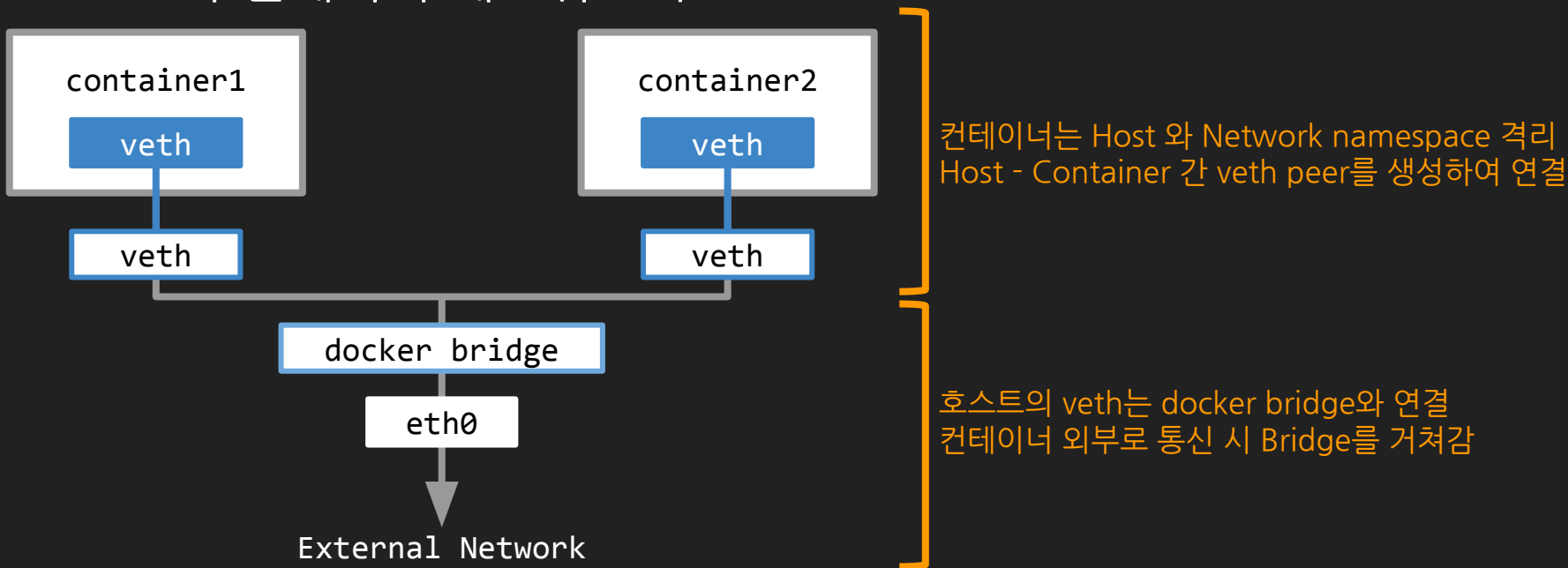
(... 생략 ...)

```
7: veth2@if8: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000  
    link/ether d2:a1:90:78:3c:4b brd ff:ff:ff:ff:ff:ff link-netnsid 0
```

# 컨테이너가 격리된 환경을 구현하는 주요 원리

## (6) Network (net) namespace

### Docker의 컨테이너 네트워크 구조



# 컨테이너가 격리된 환경을 구현하는 주요 원리

## (7) Unix Time-Sharing (uts) namespace

### Unix Time-Sharing?

컴퓨팅 자원을 다른 유저들과 공유하는 것에서 유래

여러 유저가 같은 머신을 사용하고 있지만,  
마치 다른 머신을 사용 중인 것처럼 만들고 싶은데...

# 컨테이너가 격리된 환경을 구현하는 주요 원리

## (7) Unix Time-Sharing (uts) namespace

### Unix Time-Sharing?

컴퓨팅 자원을 다른 유저들과 공유하는 것에서 유래

여러 유저가 같은 머신을 사용하고 있지만,  
마치 다른 머신을 사용 중인 것처럼 만들고 싶은데...

➡ hostname을 격리할 수 있는 공간을 만들자!

# 컨테이너가 격리된 환경을 구현하는 주요 원리

## (7) Unix Time-Sharing (uts) namespace

hostname, domainname 격리

```
$ hostname  
ubuntu
```

현재 hostname 출력

```
$ unshare -u /bin/bash
```

UTS namespace 격리한 bash 실행

```
$ hostname hyojun
```

```
$ hostname  
hyojun
```

Hostname 을 “hyojun”으로 변경 후 hostname 출력

```
$ exit
```

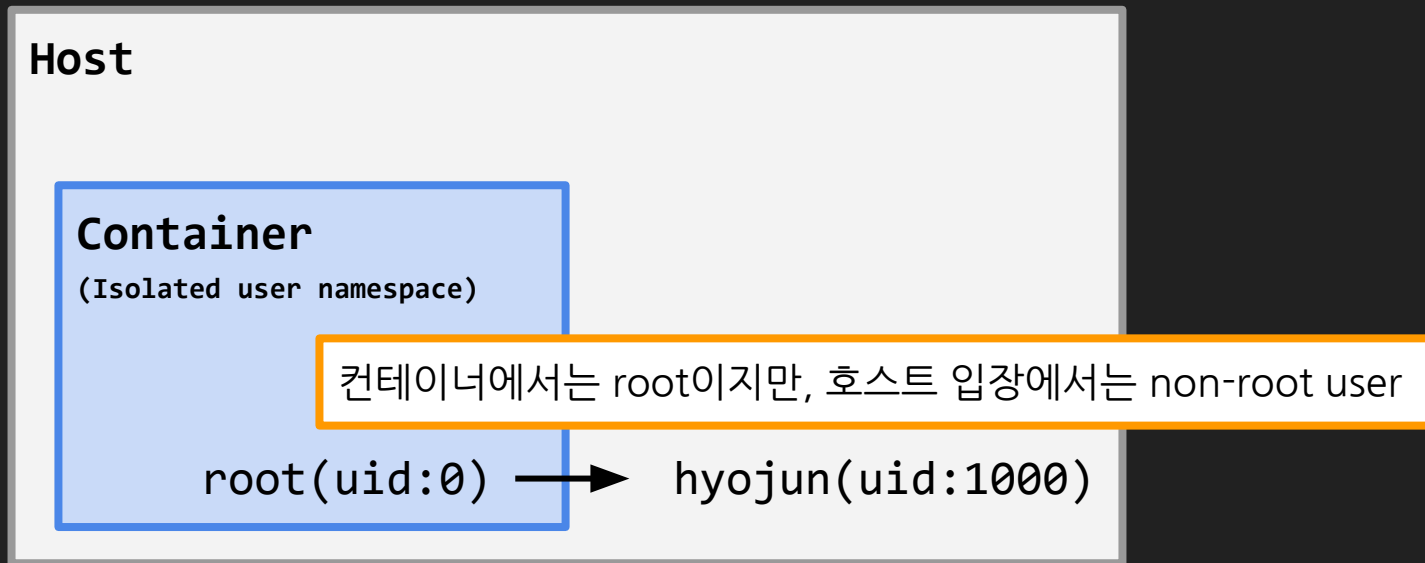
```
$ hostname  
ubuntu
```

bash 종료 후 hostname 출력 = 이전 원래의 hostname 그대로 유지  
(UTS namespace가 격리된 프로세스에서만 hostname이 변경되었음)

# 컨테이너가 격리된 환경을 구현하는 주요 원리

## (8) User ID (user) namespace

호스트에서의 uid와 Container의 uid를 다르게 매핑



# 컨테이너가 격리된 환경을 구현하는 주요 원리

## (8) User ID (user) namespace

하지만 Docker container는,  
기본적으로 user namespace를 격리하지 않는다.

호스트 PID=1의 namespace를 그대로 사용

```
06:26ubuntu@ubuntu>container> sudo lsns -p 95359
```

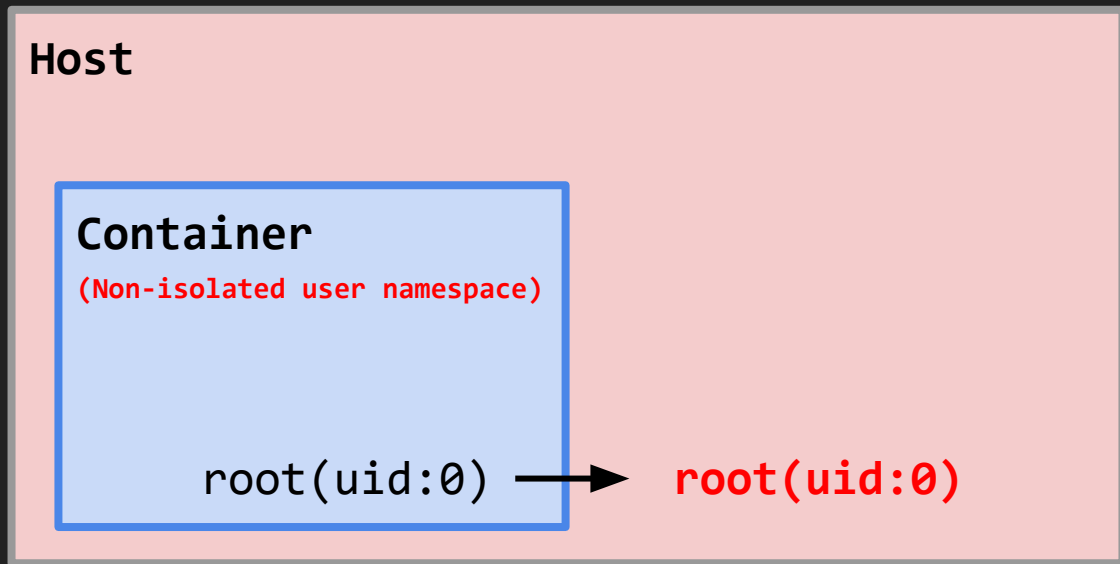
	NS TYPE	NPROCS	PID	USER	COMMAND
4026531835	cgroup	150	1	root	/sbin/init maybe-ubiquity
4026531837	user	150	1	root	/sbin/init maybe-ubiquity
4026532493	mnt	1	95359	root	sleep 3600
4026532494	uts	1	95359	root	sleep 3600
4026532495	ipc	1	95359	root	sleep 3600
4026532496	pid	1	95359	root	sleep 3600
4026532498	net	1	95359	root	sleep 3600



# 컨테이너가 격리된 환경을 구현하는 주요 원리

## (8) User ID (user) namespace

즉, 컨테이너의 user가 Host의 같은 uid 권한을 (거의) 그대로 행사할 수 있음



# 컨테이너가 격리된 환경을 구현하는 주요 원리

## (8) User ID (user) namespace

한 번쯤 Docker 설치 후 실행해봤을 명령어

```
$ sudo usermod -aG docker <your-user>
```

> Docker 설치 이후,

root가 아닌 유저가 docker를 실행할 수 있도록 docker group에 추가

# 컨테이너가 격리된 환경을 구현하는 주요 원리

## (8) User ID (user) namespace

한 번쯤 Docker 설치 후 실행해봤을 명령어

```
$ sudo docker run --user user>
```

**WARNING!**

- > Docker 설치 이후,  
root가 아닌 유저가 docker를 실행할 수 있도록 docker group에 추가

# 컨테이너가 격리된 환경을 구현하는 주요 원리

## (8) User ID (user) namespace

호스트의 “/” 루트 디렉토리 binding까지 해버리면...

```
non-root$ docker run -ti -v /:/host ubuntu:18.04 /bin/bash
```

root 권한이 없는 사용자가 Docker를 통해  
Host 디렉토리에서 root 유저 권한을 행사할 수 있습니다.

# 컨테이너가 격리된 환경을 구현하는 주요 원리

## (8) User ID (user) namespace

### 그럼 왜 Docker는 user namespace를 격리하지 않을까요?

- PID, Network namespace 공유 기능과 호환 문제
- user mapping을 지원하지 않는 외부 볼륨 또는 드라이버와의 호환 문제
- 격리된 user namespace의 user가 매핑된 실제 Host 상의 uid로부터, Host에서 binding 한 파일에 접근 권한이 보장되어야 하는 복잡성
- 격리되지 않은 username space에서, 컨테이너 root가 호스트 root와 거의 대등한 수준의 권한을 가지긴 하지만 전체 root 권한을 의미하는 건 아님.

# 컨테이너가 격리된 환경을 구현하는 주요 원리

## (8) User ID (user) namespace

**Kubernetes 또한 user namespace 격리를 아직 지원하지 않습니다**

관련해서 읽어보면 좋은 글

<https://kinvolk.io/blog/2020/12/improving-kubernetes-and-container-security-with-user-namespaces/>

# 컨테이너가 격리된 환경을 구현하는 주요 원리

## (8) User ID (user) namespace

User namespace 격리를 사용하지 않을 때는...

- 신뢰할 수 있는 사용자만 컨테이너 런타임(e.g. Docker)을 실행할 수 있도록 제한한다.
- 컨테이너의 프로세스가 root user로 실행하지 않도록 한다.
  - 특정 uid, gid로 실행될 수 있도록 지정
- 호스트의 디렉토리를 컨테이너가 직접 접근할 수 있도록 마운트 하지 않는다.

**Kubernetes에서도 같은 원리의 보안 설정을 제공합니다.**

<https://kubernetes.io/docs/concepts/policy/pod-security-policy/#users-and-groups>

# 컨테이너가 격리된 환경을 구현하는 주요 원리

## (9) Control group (cgroup)

프로세스 그룹의 자원 할당을 제한하고 격리할 수 있는 리눅스 커널 기능

- CPU
- Memory
- Network
- Disk

CPU 사용량을 제한하거나...  
메모리 사용량을 제한하고...  
네트워크 트래픽 우선순위를 설정하거나...  
사용량에 대한 통계를 제공하는 등



# 요약: 컨테이너가 격리된 환경을 구현하는 주요 원리

- 컨테이너는 “격리된 환경”에서 실행되는 “프로세스”이다.
- namespace를 통해 프로세스의 “격리된 환경”을 구현
  - Mount (mnt)
  - Process ID (pid)
  - Network (net)
  - Interprocess Communication (ipc)
  - Unix Time-Sharing (uts)
  - User ID (user)
- cgroups을 통해 프로세스의 자원 사용량을 제한

# What is Kubernetes Pod?

- 쿠버네티스에서 배포할 수 있는 최소 객체 단위
- 1개 이상의 컨테이너로 이루어진 그룹

K8s application은 Pod 단위로 배포됩니다.

## Kubernetes Cluster

Node 1

Node 2

...

K8s application은 Pod 단위로 배포됩니다.

replicas: **1**



## Kubernetes Cluster

Node 1

Node 2

...

K8s application은 Pod 단위로 배포됩니다.

replicas: **1**



## Kubernetes Cluster

Node 1



Node 2

...

K8s application은 Pod 단위로 배포됩니다.

replicas: 2



## Kubernetes Cluster

Node 1



Node 2



...

# Pod은 “배포 가능한 최소 객체 단위”?

- 실제로 Pod 은 여러 형태의 리소스에 의해 배포됩니다.
  - Job - 한 번 실행되고 작업이 완료되면 종료되는 형태의 Pod을 관리
  - ReplicaSet - 명시된 Pod 개수를 실행되는 상태를 보장(replica)
  - DaemonSet - 각 노드마다 하나씩만 실행되는 Pod을 관리
  - StatefulSet - Stateful application을 실행하는 Pod을 관리
  - Deployment - Pod, ReplicaSet의 업데이트에 대한 배포를 관리

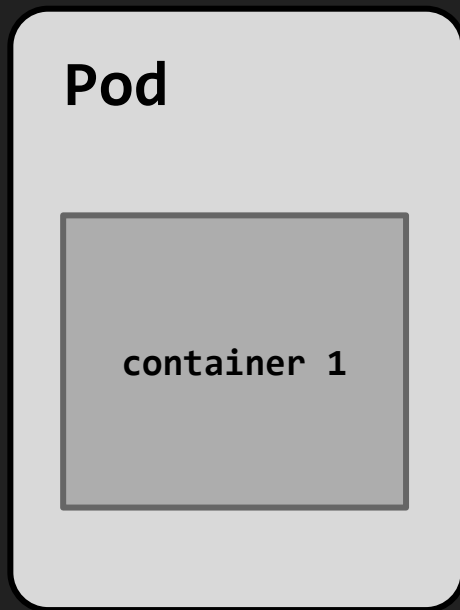
결국 Pod은 Kubernetes에서 생성하고 관리되는 가장 기본적인 단위

# What is Kubernetes Pod?

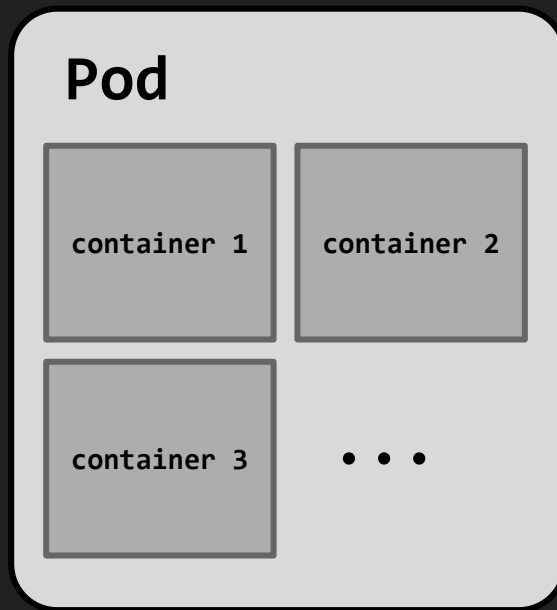
- 쿠버네티스에서 배포할 수 있는 최소 객체 단위
- 1개 이상의 컨테이너로 이루어진 그룹



Pod에는 1개 이상의 컨테이너가 존재할 수 있습니다.

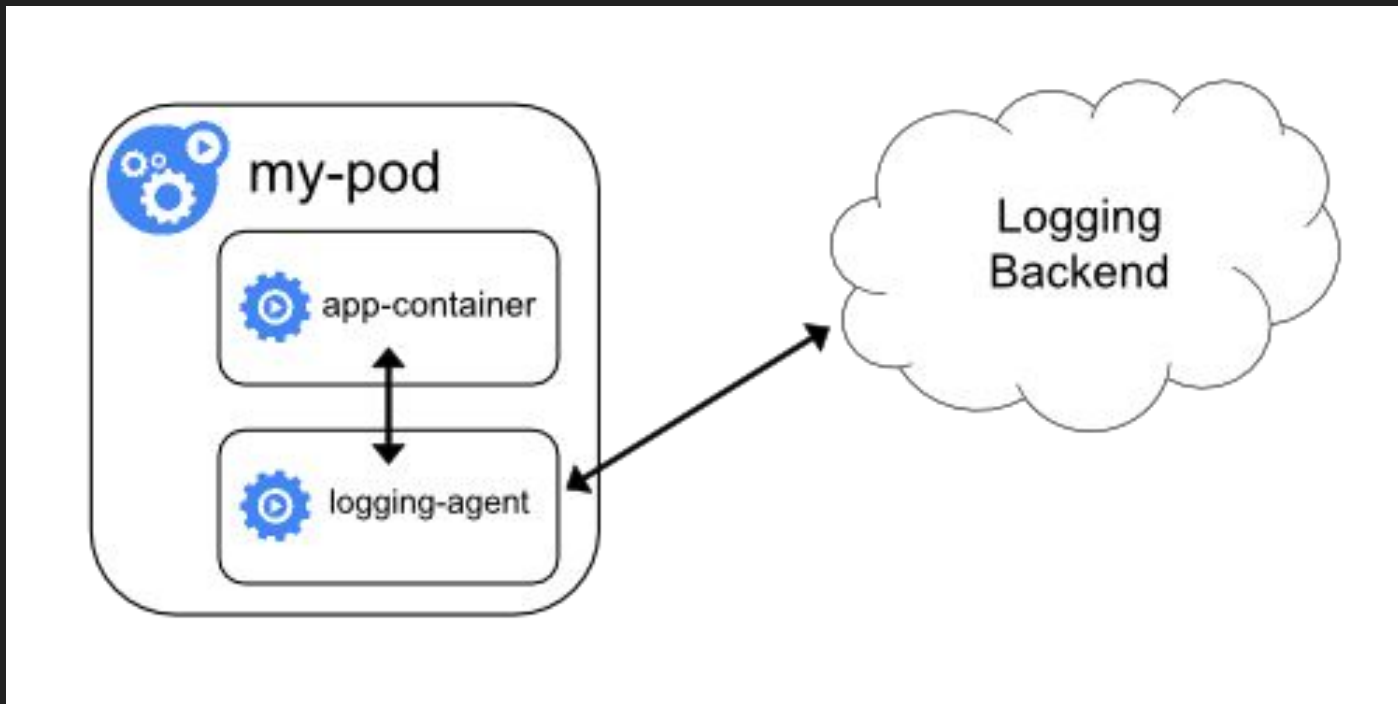


단일 컨테이너를 실행하는 pod



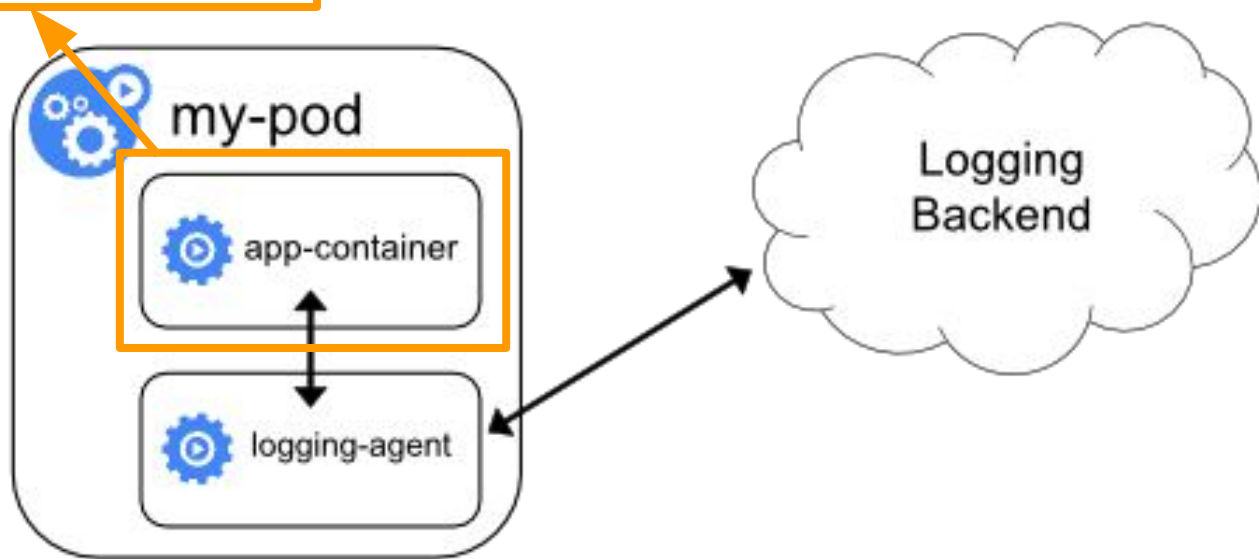
여러 컨테이너를 실행하는 pod

# 여러 컨테이너를 실행하는 Pod의 사례

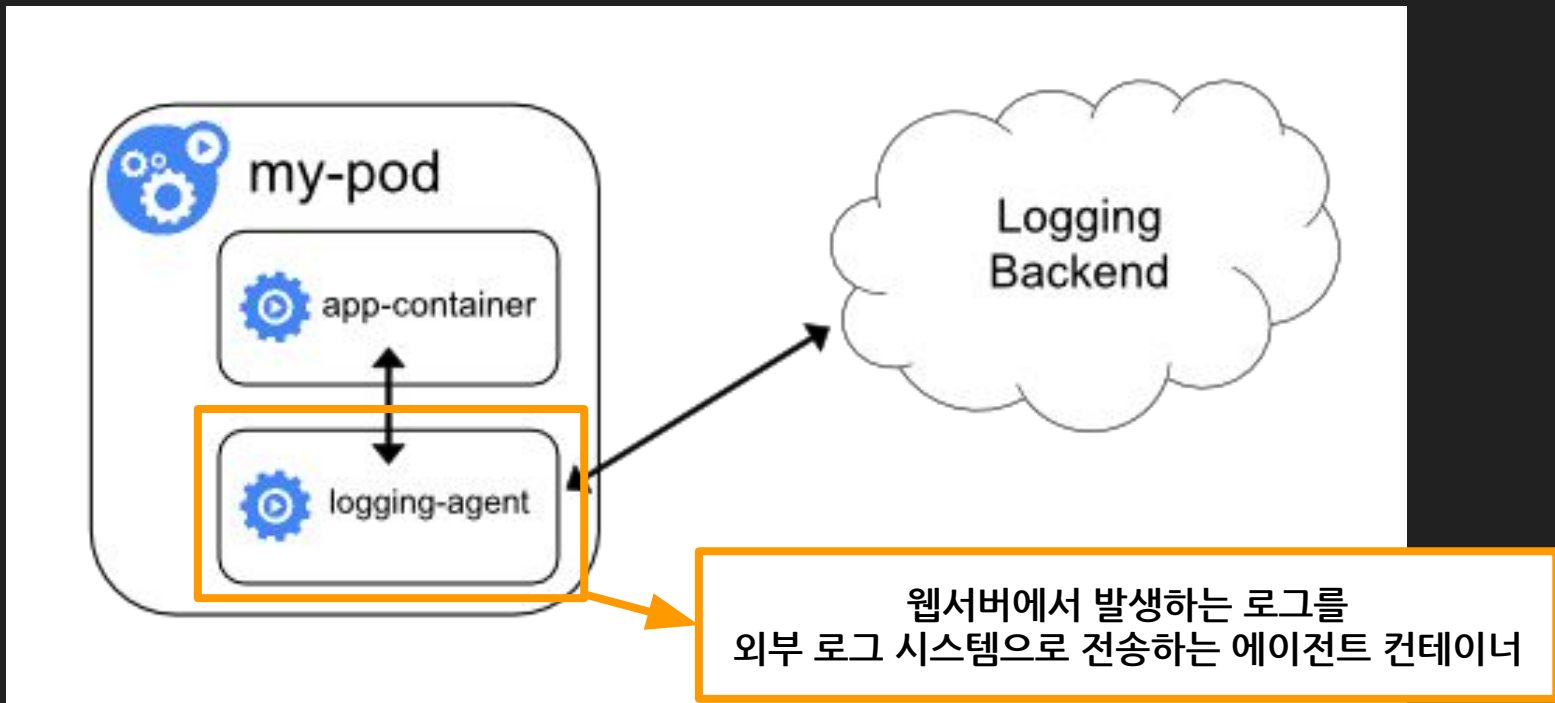


# 여러 컨테이너를 실행하는 Pod의 사례

웹서버를 실행하는 컨테이너



# 여러 컨테이너를 실행하는 Pod의 사례



# Pod이 여러 컨테이너로 구성하는 경우

- 주 역할을 하는 1개의 **Primary Container**
- 1개 이상의 **Sidecar Containers**
  - 주 컨테이너의 보완 역할을 하기 위해 실행되는 컨테이너  
예) 모니터링, 로깅 등 ...

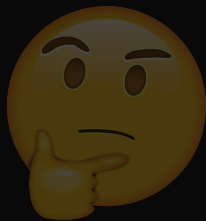


오토바이에 연결된 “Sidecar”



한 컨테이너에서 모두 실행하면 안 되나요?

굳이 별도 컨테이너로 분리하지 말고...  
복잡한데...



한 컨테이너에서 두 실행하면 안 되나요?



굳이 별도 컨테이너로 분리하지 말고...  
복잡한데...

할 순 있지만, 권장하지 않습니다.

# 컨테이너에서는 단일 프로세스를 실행하도록 권장

- 컨테이너는 “격리된 환경”에서 실행되는 “프로세스”임을 배웠습니다.
- 격리된 PID namespace에서 최초로 실행된 프로세스는 pid=1 입니다.

**컨테이너에서 최초로 실행된 프로세스의 상태**

**=**

**컨테이너의 수명**



# 컨테이너 안에서 여러 프로세스가 실행 중이라면?

컨테이너가 실행 중이라도,  
메인 프로세스를 제외한 다른 프로세스들이 실행 중인 상태를 보장할 수 없음.

**컨테이너에서 실행되는 프로세스들의 상태**

**≠**

**컨테이너의 상태**

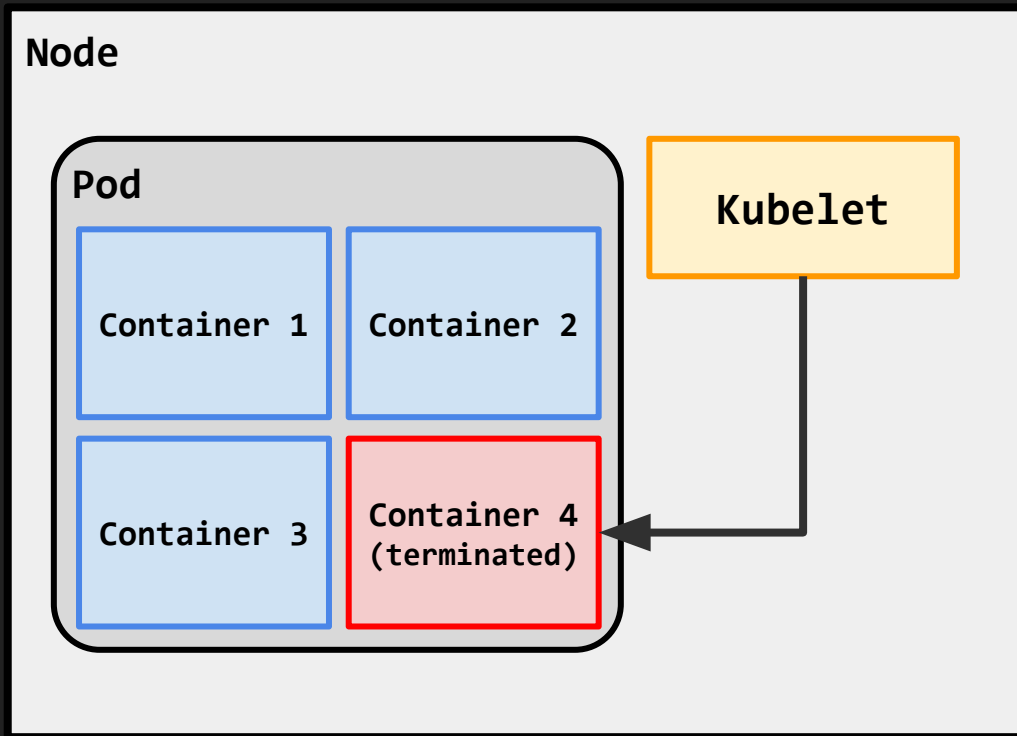
# Kubernetes Pod의 특정 Container가 종료되면?

```
spec:
  template:
    (...)
    spec:
      containers:
        - name: hello
          image: busybox
          command: ['sh', '-c', 'sleep 3600']
          restartPolicy: Always
    (...)
  (...)
  (...)
```

걱정 마세요, Kubernetes가 선언된 **restartPolicy**에 따라 컨테이너를 재시작합니다.

- Always
- OnFailure
- Never

# Kubernetes Pod의 특정 Container가 종료되면?



exponential back-off delay마다 재시도  
(10s, 20s, 40s, ... 최대 5분까지)

# Pod를 구성하는 기준

- 컨테이너들이 꼭 같은 노드에서 실행되어야 하는가?  
(같은 Pod에 존재하는 컨테이너들은 항상 같은 노드에 존재)
- 해당 컨테이너들이 같은 개수로 수평 확장되어야 하는가?  
(Pod 단위는 곧 확장의 단위)
- 컨테이너들을 하나의 그룹으로 함께 배포해야 하는가?

# Kubernetes Pod의 컨테이너간 격리

- Pod은 1개 이상의 컨테이너로 이루어진 그룹
- 그렇다면... 같은 Pod의 컨테이너간 격리는 어떻게 이루어 질까?

# Kubernetes Pod의 컨테이너간 격리

- Pod은 1개 이상의 컨테이너로 이루어진 그룹
- 그렇다면... 같은 Pod의 컨테이너간 격리는 어떻게 이루어 질까?

컨테이너의 원리를 알았으니...  
직접 들여다봅시다!

# Kubernetes Pod의 컨테이너간 격리

## two-containers-pod.yml

```
apiVersion: batch/v1
kind: Job
metadata:
  name: two-containers-pod
spec:
  template:
    # This is the pod template
    spec:
      containers:
        - name: hello
          image: busybox
          command: ['sh', '-c', 'echo "first container" && sleep 3600']
        - name: hello2
          image: busybox
          command: ['sh', '-c', 'echo "second container" && sleep 3600']
      restartPolicy: OnFailure
    # The pod template ends here
```

} 2개 컨테이너가 실행되는 pod

# Kubernetes Pod의 컨테이너간 격리

(Kubernetes v1.20.2)

```
vagrant@control-plane:~$ kubectl apply -f two-containers-pod.yml  
job.batch/two-containers-pod created
```

```
vagrant@control-plane:~$ kubectl get pods -o wide
```

NAME	READY	STATUS	RESTARTS	AGE	IP	NODE
two-containers-pod-9pvvt	2/2	Running	0	20s	10.244.2.4	worker-node2

Pod이 실행 중인 노드에서 컨테이너를 확인해보면...



# Kubernetes Pod의 컨테이너간 격리

```
vagrant@worker-node2:~$ ps -ef | grep sleep
root      9318   9295   0 20:01 ?        00:00:00 sleep 3600
root      9382   9356   0 20:01 ?        00:00:00 sleep 3600
vagrant  10647  10383   0 20:04 pts/0    00:00:00 grep --color=auto sleep
```

```
vagrant@worker-node2:~$ sudo lsns -p 9318
```

NS	TYPE	NPROCS	PID	USER	COMMAND
4026531835	cgroup	121	1	root	/sbin/init
4026531837	user	121	1	root	/sbin/init
4026532214	ipc	3	9182	root	/pause
4026532217	net	3	9182	root	/pause
4026532304	mnt	1	9318	root	sleep 3600
4026532305	uts	1	9318	root	sleep 3600
4026532306	pid	1	9318	root	sleep 3600

```
vagrant@worker-node2:~$ sudo lsns -p 9382
```

NS	TYPE	NPROCS	PID	USER	COMMAND
4026531835	cgroup	121	1	root	/sbin/init
4026531837	user	121	1	root	/sbin/init
4026532214	ipc	3	9182	root	/pause
4026532217	net	3	9182	root	/pause
4026532307	mnt	1	9382	root	sleep 3600
4026532308	uts	1	9382	root	sleep 3600
4026532309	pid	1	9382	root	sleep 3600

cgroup namespace, user namespace는 따로 격리하지 않음

\*cgroup namespace

# Kubernetes Pod의 컨테이너간 격리

```
vagrant@worker-node2:~$ ps -ef | grep sleep
root      9318   9295   0 20:01 ?        00:00:00 sleep 3600
root      9382   9356   0 20:01 ?        00:00:00 sleep 3600
vagrant  10647  10383   0 20:04 pts/0    00:00:00 grep --color=auto sleep
```

```
vagrant@worker-node2:~$ sudo lsns -p 9318
      NS TYPE  NPROCS   PID USER COMMAND
4026531835 cgroup    121     1 root /sbin/init
4026531837 user     121     1 root /sbin/init
4026532214 ipc        3   9182 root /pause
4026532217 net        3   9182 root /pause
4026532304 mnt         1   9318 root sleep 3600
4026532305 uts         1   9318 root sleep 3600
4026532306 pid         1   9318 root sleep 3600
```

```
vagrant@worker-node2:~$ sudo lsns -p 9382
      NS TYPE  NPROCS   PID USER COMMAND
4026531835 cgroup    121     1 root /sbin/init
4026531837 user     121     1 root /sbin/init
4026532214 ipc        3   9182 root /pause
4026532217 net        3   9182 root /pause
4026532307 mnt         1   9382 root sleep 3600
4026532308 uts         1   9382 root sleep 3600
4026532309 pid         1   9382 root sleep 3600
```

mnt, uts, pid namespace는 각 컨테이너별로 격리  
(같은 pod이라도 공유하지 않음)

# Kubernetes Pod의 컨테이너간 격리

```
vagrant@worker-node2:~$ ps -ef | grep sleep
root      9318   9295   0 20:01 ?        00:00:00 sleep 3600
root      9382   9356   0 20:01 ?        00:00:00 sleep 3600
vagrant  10647  10383   0 20:04 pts/0    00:00:00 grep --color=auto sleep
```

```
vagrant@worker-node2:~$ sudo lsns -p 9318
      NS TYPE  NPROCS  PID USER COMMAND
4026531835 cgroup    121     1 root /sbin/init
4026531837 user     121     1 root /sbin/init
```

```
4026532214 ipc        3   9182 root /pause
4026532217 net        3   9182 root /pause
4026532304 mnt         1   9318 root sleep 3600
4026532305 uts         1   9318 root sleep 3600
4026532306 pid         1   9318 root sleep 3600
```

```
vagrant@worker-node2:~$ sudo lsns -p 9382
      NS TYPE  NPROCS  PID USER COMMAND
4026531835 cgroup    121     1 root /sbin/init
4026531837 user     121     1 root /sbin/init
```

```
4026532214 ipc        3   9182 root /pause
4026532217 net        3   9182 root /pause
4026532307 mnt         1   9382 root sleep 3600
4026532308 uts         1   9382 root sleep 3600
4026532309 pid         1   9382 root sleep 3600
```

ipc, net namespace는 pod의 컨테이너 간 공유

→ 컨테이너 프로세스 간 공유 메모리 등의 IPC 가능

→ 컨테이너 간 동일한 IP 주소, 포트를 공유(충돌 주의)

## Pause?

# Pause Container?

```
vagrant@worker-node2:~$ docker ps | grep two-containers
ae2b5c6f7882          busybox               "sh -c 'echo \"second...\"
e8670003df44          busybox               "sh -c 'echo \"first ...\"
19802e369987          k8s.gcr.io/pause:3.2  "/pause"
```

Pause 컨테이너는 격리된 IPC, Network namespace를 생성하고 유지  
→ 나머지 컨테이너들은 해당 namespace를 공유하여 사용

유저가 실행한 특정 컨테이너가 비정상 종료되어,  
컨테이너 전체에서 공유되는 namespace에 문제가 발생하는 것을 방지

# Pause Container?

1. 단순히 무한 루프를 돌면서,  
SIGINT, SIGTERM을 받으면 종료

```
32 static void sigdown(int signo) {
33     psignal(signo, "Shutting down, got signal");
34     exit(0);
35 }
36
37 static void sigreap(int signo) {
38     while (waitpid(-1, NULL, WNOHANG) > 0)
39         ;
40 }
41
42 int main(int argc, char **argv) {
43     int i;
44     for (i = 1; i < argc; ++i) {
45         if (!strcasecmp(argv[i], "-v")) {
46             printf("pause.c %s\n", VERSION_STRING(VERSION));
47             return 0;
48         }
49     }
50
51     if (getpid() != 1)
52         /* Not an error because pause sees use outside of infra containers. */
53         fprintf(stderr, "Warning: pause should be the first process\n");
54
55     if (sigaction(SIGINT, &(struct sigaction){.sa_handler = sigdown, NULL}) < 0)
56         return 1;
57     if (sigaction(SIGTERM, &(struct sigaction){.sa_handler = sigdown, NULL}) < 0)
58         return 2;
59     if (sigaction(SIGCHLD, &(struct sigaction){.sa_handler = sigreap,
60                                                .sa_flags = SA_NOCLDSTOP},
61                 NULL) < 0)
62         return 3;
63
64     for (;;)
65         pause();
66     fprintf(stderr, "Error: infinite loop terminated\n");
67     return 42;
68 }
```

# Pause Container?

1. 단순히 무한 루프를 돌면서,  
SIGINT, SIGTERM을 받으면 종료
2. Zombie Process Reaping 역할  
(PID namespace sharing 하는 경우)

```
42 static void sigdown(int signo) {
43     psignal(signo, "Shutting down, got signal");
44     exit(0);
45 }
46
47 static void sigreap(int signo) {
48     while (waitpid(-1, NULL, WNOHANG) > 0)
49         ;
50 }
51
52 int main(int argc, char **argv) {
53     int i;
54     for (i = 1; i < argc; ++i) {
55         if (!strcasecmp(argv[i], "-v")) {
56             printf("pause.c %s\n", VERSION_STRING(VERSION));
57             return 0;
58         }
59     }
60
61     if (getpid() != 1)
62         /* Not an error because pause sees use outside of infra containers. */
63         fprintf(stderr, "Warning: pause should be the first process\n");
64
65     if (sigaction(SIGINT, &(struct sigaction){.sa_handler = sigdown, NULL}) < 0)
66         return 1;
67     if (sigaction(SIGTERM, &(struct sigaction){.sa_handler = sigdown, NULL}) < 0)
68         return 2;
69     if (sigaction(SIGCHLD, &(struct sigaction){.sa_handler = sigreap,
70                                                 .sa_flags = SA_NOCLDSTOP,
71                                                 NULL}) < 0)
72         return 3;
73
74     for (;;)
75         pause();
76     fprintf(stderr, "Error: infinite loop terminated\n");
77     return 42;
78 }
```

# Kubernetes의 PID namespace sharing

각 컨테이너에서 Zombie process가 발생할 우려가 있는 경우,  
Kubernetes “PID namespace sharing” 옵션을 활성화하여  
Pause 컨테이너에 Zombie process reaping 역할을 위임할 수 있다.

참고 링크: “**Zombie process reaping에 대하여, Container에서 고려할 부분들**”  
→ <https://blog.hyojun.me/4>

- Kubernetes v1.7에서는 PID namespace sharing이 기본적으로 활성화
- 하지만 v1.8부터는 기존 init system에 의존하는 컨테이너와의 호환성 이슈로 다시 비활성화됨  
<https://github.com/kubernetes/kubernetes/issues/48937>



# Kubernetes의 PID namespace sharing

two-containers-pod.yml

```
apiVersion: batch/v1
kind: Job
metadata:
  name: two-containers-pod
spec:
  template:
    # This is the pod template
    spec:
      shareProcessNamespace: true
      containers:
        - name: hello
          image: busybox
          command: ['sh', '-c', 'echo "first container" && sleep 3600']
        - name: hello2
          image: busybox
          command: ['sh', '-c', 'echo "second container" && sleep 3600']
      restartPolicy: OnFailure
    # The pod template ends here
```

요 설정만 추가해 줍니다.



# Kubernetes의 PID namespace sharing

```
vagrant@worker-node1:~$ ps -ef | grep sleep
root      1009   983   0  21:01 ?        00:00:00 sleep 3600
root      1083  1050   0  21:01 ?        00:00:00 sleep 3600
vagrant   1378  1297   0  21:01 pts/0    00:00:00 grep --color=auto sleep
```

```
vagrant@worker-node1:~$ sudo lsns -p 1009
      NS TYPE      NPROCS   PID USER COMMAND
4026531835 cgroup    117      1 root /sbin/init
4026531837 user      117      1 root /sbin/init
4026532214 ipc         3    877 root /pause
4026532215 pid         3    877 root /pause
4026532217 net         3    877 root /pause
4026532304 mnt         1   1009 root sleep 3600
4026532305 uts         1   1009 root sleep 3600
```

pause 컨테이너의 PID namespace

```
vagrant@worker-node1:~$ docker exec e077f522d6fa ps
```

PID	USER	TIME	COMMAND
1	root	0:00	/pause
6	root	0:00	sleep 3600
11	root	0:00	sleep 3600
22	root	0:00	ps

- sleep 프로세스를 실행하는 컨테이너에서, 같은 pod의 다른 컨테이너 프로세스들을 볼 수 있다.
- 같은 pod에서는 pid 1은 항상 pause 프로세스이다.

# Kubernetes 없이 Pod 만들어 보기(실습)

```
# Docker version 19.03.15
```

```
# Pause 컨테이너 생성
```

```
$ docker run -d --ipc="shareable" --name pause k8s.gcr.io/pause:3.2
```

```
# Pause 컨테이너의 ipc, net, pid namespace를 공유하는 컨테이너 생성
```

```
$ docker run -ti --rm -d --name sleep-busybox \  
    --net=container:pause \  
    --ipc=container:pause \  
    --pid=container:pause \  
    busybox sleep 3600
```

```
# 공유된 namespace에서 ps 명령어를 실행하는 컨테이너 생성
```

```
# 서로 다른 컨테이너이지만 같은 pid namespace를 공유하는 pause, sleep 컨테이너의 프로세스가 보인다.
```

```
$ docker run --rm --name ps-busybox \  
    --net=container:pause \  
    --ipc=container:pause \  
    --pid=container:pause \  
    busybox ps
```

```
# sleep-busybox 컨테이너의 namespace를 살펴본다.
```

```
$ ps -ef | grep sleep
```

```
$ sudo lsns -p <PID>
```

# Kubernetes Pod과 비교(실습)

## practice.yml

```
apiVersion: batch/v1
kind: Job
metadata:
  name: practice
spec:
  template:
    # This is the pod template
    spec:
      shareProcessNamespace: true
      containers:
        - name: sleep
          image: busybox
          command: ['sh', '-c', 'sleep 3600']
      restartPolicy: OnFailure
    # The pod template ends here
```

```
$ kubectl apply -f practice.yml
```

```
# 어떤 노드에 실행되고 있는지 확인
```

```
$ kubectl get pods -o wide
```

```
# 해당 노드에서 sleep 컨테이너의 namespace를 살펴본다.
```

```
node# $ ps -ef | grep sleep
```

```
node# $ sudo lsns -p <PID>
```

# 요약: Kubernetes Pod의 개념

- Pod이란?
  - 쿠버네티스에서 배포할 수 있는 최소 객체 단위
    - Pod은 여러 형태의 리소스에 의해 배포(Job, ReplicaSet, 등)
  - 1개 이상의 컨테이너로 이루어진 그룹
    - 단일 컨테이너를 실행하는 Pod
    - 여러 컨테이너를 실행하는 Pod
      - Primary Container
      - Sidecar Containers

# 요약: Kubernetes Pod의 개념

- 한 컨테이너에서 여러 프로세스를 실행하는 것은 권장하지 않음
  - 컨테이너가 실행 중이라도,  
메인 프로세스를 제외한 다른 프로세스들이 실행 중인 상태를 보장할 수 없음.
- Kubernetes Pod의 특정 컨테이너가 종료되면,  
Kubelet이 restartPolicy에 따라 컨테이너를 재시작한다.
- Pod을 어떻게 구성할지 판단 기준
  - 컨테이너들이 꼭 같은 노드에서 실행되어야 하는가?
  - 해당 컨테이너들이 같은 개수로 수평 확장되어야 하는가?
  - 컨테이너들을 하나의 그룹으로 함께 배포해야 하는가?

# 요약: Kubernetes Pod의 개념

- Pod의 컨테이너 간 격리
  - Host와 공유되는 namespace → **cgroup, user**
  - 같은 Pod의 컨테이너 간 공유되는 namespace → **ipc, net**
  - 컨테이너 별로 격리되는 namespace → **mount, uts, pid**
    - pid namespace 공유는 optional
- Pause Container?
  - 컨테이너 간 공유될 IPC, Network namespace를 생성하고 유지
  - PID namespace 공유 시 Zombie process reaping 역할도 수행

감사합니다.