실습과 함께 완성해보는

도커 없이 컨테이너 만들기

8편

Sam.0

실습을 위한 사전 준비 사항

실습 환경 구성하기 클릭

실습은 . . .

- 맥 환경에서 VirtualBox + Vagrant 기반으로 준비되었습니다
  - 맥 이외의 OS 환경도 괜찮습니다만
  - 원활한 실습을 위해서
  - “VirtualBox or VMware + Vagrant”는 권장드립니다.
- 실습환경 구성을 위한 Vagrantfile을 제공합니다.

실습을 위한 사전 준비 사항

Vagrantfile

- 오른쪽의 텍스트를 복사하신 후
- 로컬에 Vagrantfile로 저장하여 사용하면 됩니다.
- vagrant 사용법은 공식문서를 참고해 주세요 https://www.vagrantup.com/docs/index

실습 환경 구성하기 클릭

```
BOX_IMAGE = "bento/ubuntu-18.04"
HOST_NAME = "ubuntu1804"

$pre_install = <<-SCRIPT
  echo ">>>> pre-install <<<<<<"
  sudo apt-get update &&
  sudo apt-get -y install gcc &&
  sudo apt-get -y install make &&
  sudo apt-get -y install pkg-config &&
  sudo apt-get -y install libseccomp-dev &&
  sudo apt-get -y install tree &&
  sudo apt-get -y install jq &&
  sudo apt-get -y install bridge-utils

  echo ">>>> install go <<<<<<"
  curl -O https://storage.googleapis.com/golang/go1.15.7.linux-amd64.tar.gz > /dev/null 2>&1 &&
  tar xf go1.15.7.linux-amd64.tar.gz &&
  sudo mv go /usr/local/ &&
  echo 'PATH=$PATH:/usr/local/go/bin' | tee /home/vagrant/.bash_profile

  echo ">>>>> install docker <<<<<<"
  sudo apt-get -y install apt-transport-https ca-certificates curl gnupg-agent software-properties-common > /dev/null 2>&1 &&
  sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add - &&
  sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable" &&
  sudo apt-get update &&
  sudo apt-get -y install docker-ce docker-ce-cli containerd.io > /dev/null 2>&1
SCRIPT

Vagrant.configure("2") do |config|

 config.vm.define HOST_NAME do |subconfig|
   subconfig.vm.box = BOX_IMAGE
   subconfig.vm.hostname = HOST_NAME
   subconfig.vm.network :private_network, ip: "192.168.104.2"
   subconfig.vm.provider "virtualbox" do |v|
     v.memory = 1536
     v.cpus = 2
   end
   subconfig.vm.provision "shell", inline: $pre_install
 end

end
```

실습을 위한 사전 준비 사항

실습 환경 구성하기 클릭

Vagrantfile 제공 환경

vagrant + virtual vm
ubuntu 18.04
docker 20.10.5 ** 도커 이미지 다운로드 및 컨테이너 비교를 위한 용도로 사용합니다.*

기타 설치된 툴
~ tree, jq, brctl, … 등 실습을 위한 툴

실습 계정 (root)

```
# sudo -Es
```

실습 폴더

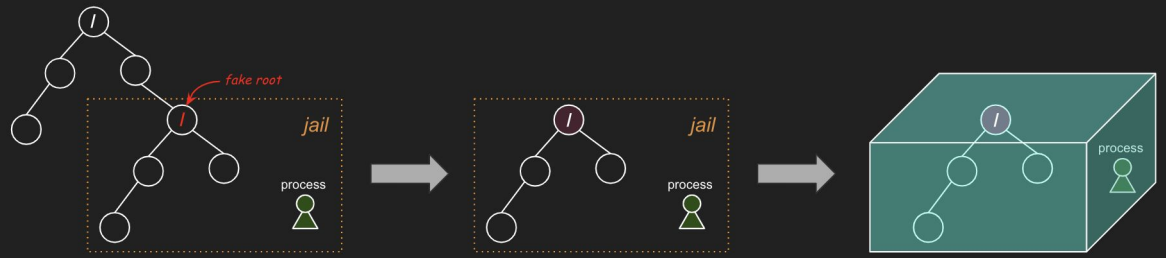
```
# cd /tmp
```

초기의 컨테이너

root isolation + mount isolation 으로 충분하다고 생각

컨테이너 ?

per-process root (/) filesystem isolation



chroot

pivot_root

mount namespace

아니 근데... ?

호스트의 프로세스들이 다 보이네 ...

container

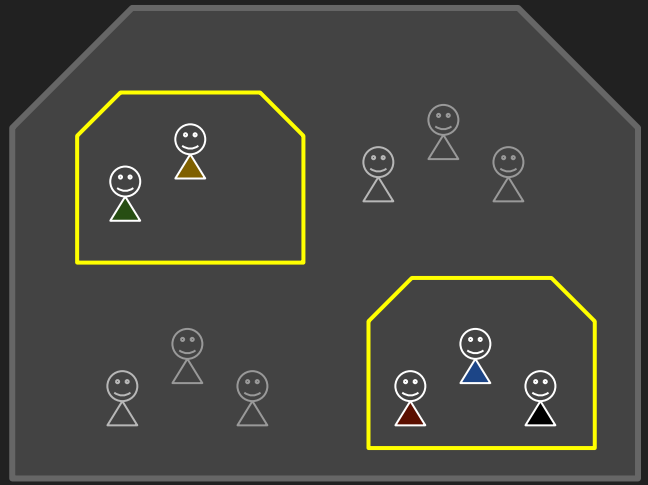
```
bash-4.4# ps aux
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
0            1  0.0  0.5  77708  8648 ?        Ss   May04   0:02 /sbin/init
0            2  0.0  0.0      0     0 ?        S    May04   0:00 [kthreadd]
0            4  0.0  0.0      0     0 ?        I<   May04   0:00 [kworker/0:0H]
0            6  0.0  0.0      0     0 ?        I<   May04   0:00 [mm_percpu_wq]
0            7  0.0  0.0      0     0 ?        S    May04   0:01 [ksoftirqd/0]
0            8  0.0  0.0      0     0 ?        I    May04   0:04 [rcu_sched]
0            9  0.0  0.0      0     0 ?        I    May04   0:00 [rcu_bh]
0           10  0.0  0.0      0     0 ?        S    May04   0:00 [migration/0]
0           11  0.0  0.0      0     0 ?        S    May04   0:01 [watchdog/0]
0           12  0.0  0.0      0     0 ?        S    May04   0:00 [cpuhp/0]
0           13  0.0  0.0      0     0 ?        S    May04   0:00 [cpuhp/1]
0           14  0.0  0.0      0     0 ?        S    May04   0:00 [watchdog/1]
0           15  0.0  0.0      0     0 ?        S    May04   0:00 [migration/1]
0           16  0.0  0.0      0     0 ?        S    May04   0:00 [ksoftirqd/1]
0           18  0.0  0.0      0     0 ?        I<   May04   0:00 [kworker/1:0H]
0           19  0.0  0.0      0     0 ?        S    May04   0:00 [kdevtmpfs]
0           20  0.0  0.0      0     0 ?        I<   May04   0:00 [netns]
0           21  0.0  0.0      0     0 ?        S    May04   0:00 [rcu_tasks_kthre]
```

pid namespace

내 것만 딱 ~ 보일 수는 없을까요?

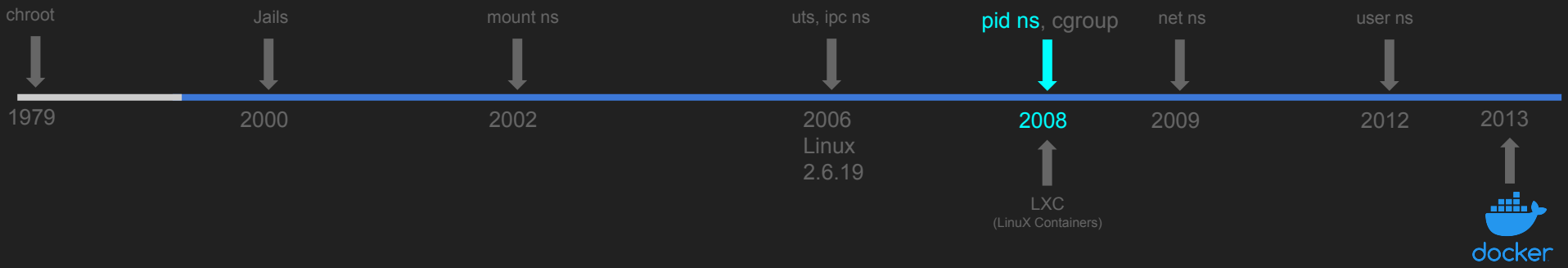
container 이렇게 ~

```
USER         PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root           1  0.0  0.2  21480  3904 pts/1    S    03:01   0:00 -bash
root           6  0.0  0.2  37376  3108 pts/1    R+   03:01   0:00 ps aux
```



PID Namespace

PID Namespace : isolates Process IDs

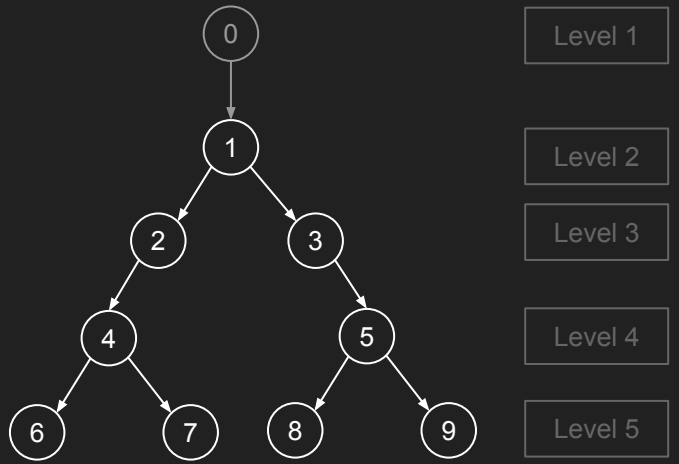


pid ?

- process identifier
- unique
- from 1

process tree ?

- tree-like structure
- parent --(fork)--> child
- tracking process



Q) if process dies ?

process dies?

child process가 죽으면?

- x (parent)
- y, z (child)
- parent --(wait)-- child

Process Table			
PID1	item		
PID2	item		
...	...		
PIDm	item		
...	...		
PIDx	PPIDm	details	exit status
...	...		
...	...		

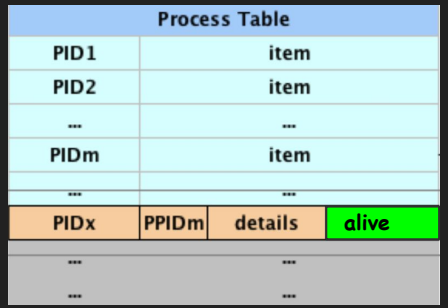
process dies? zombie

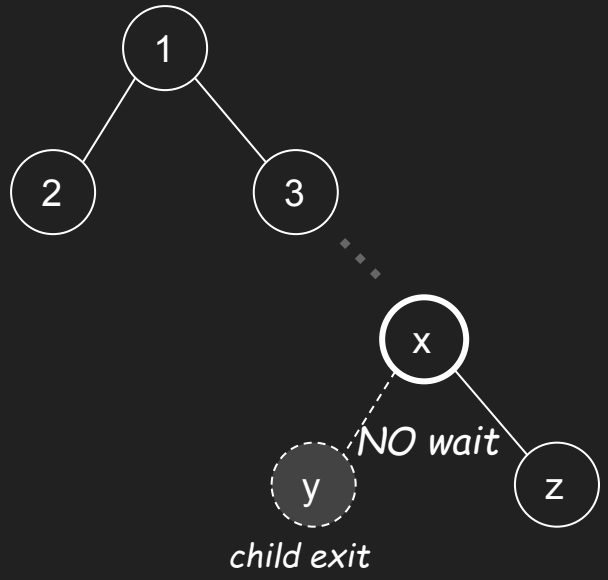
부모가 wait하지 않으면 ?

- parent NO wait ?
- Zombie *(서류상으로만 남아있음)*

  ~ 부모가 종료할 때까지 남아 있게 됨

  (부모가 아주 오래 오래 ~ 살면? ㅋㅋ)

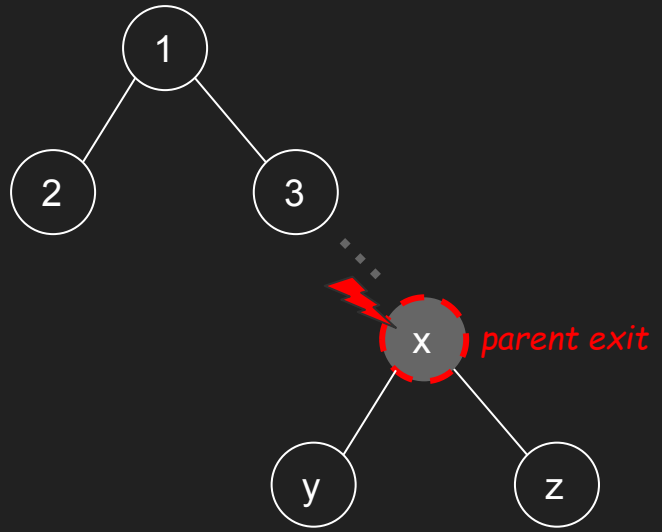




process dies?

부모 프로세스가 *먼저* 죽으면?

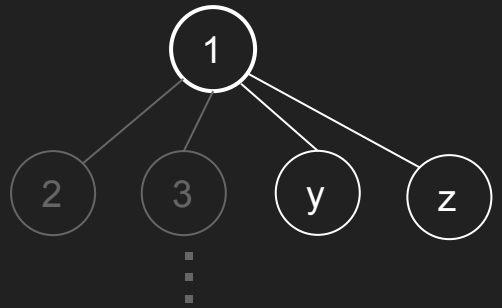
- x (parent)
- y, z (child)



process dies? orphan

부모 프로세스가 *먼저* 죽으면?

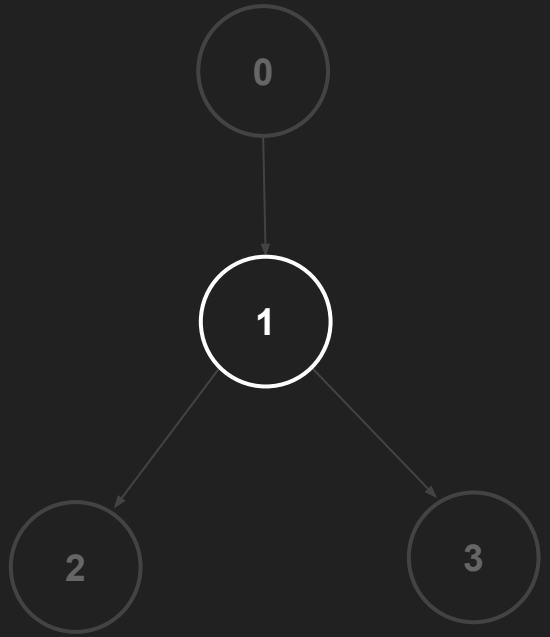
- ~~x (parent)~~ ~ *die*
- y, z (child) ~ orphan
- pid 1 (new parent)



pid 1 ~ orphan(y,z)을 거두고 종료되면 정리 (reaping)

pid 1 ? 난 특별 ~

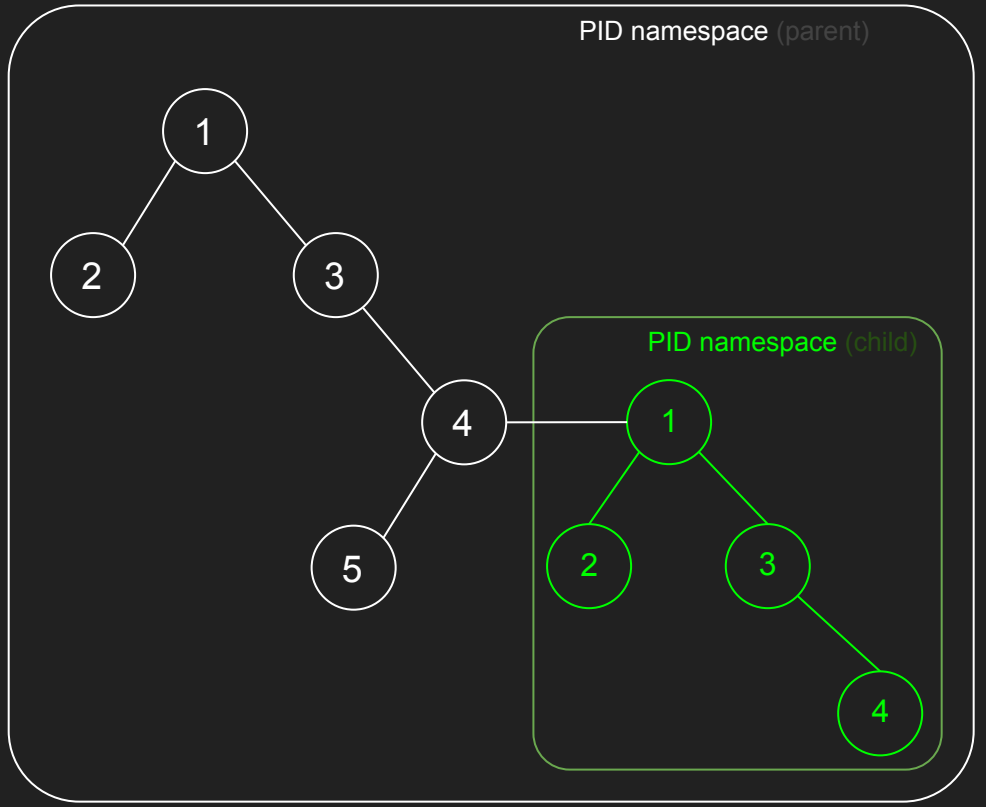
- init process
- created by kernel (pid 0)
- root of process tree (in user-space)
- reaping zombie, orphan
- if dies → panic (reboot required)



pid namespace

isolate the process ID number space

- pid number space
- unique view for application
- has own process tree

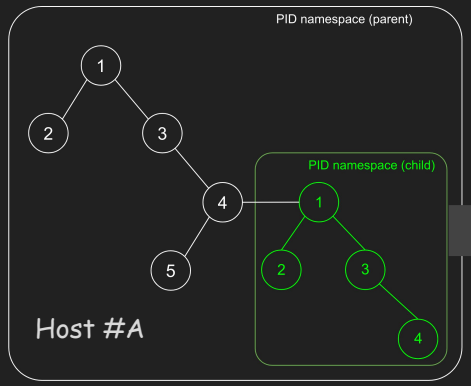


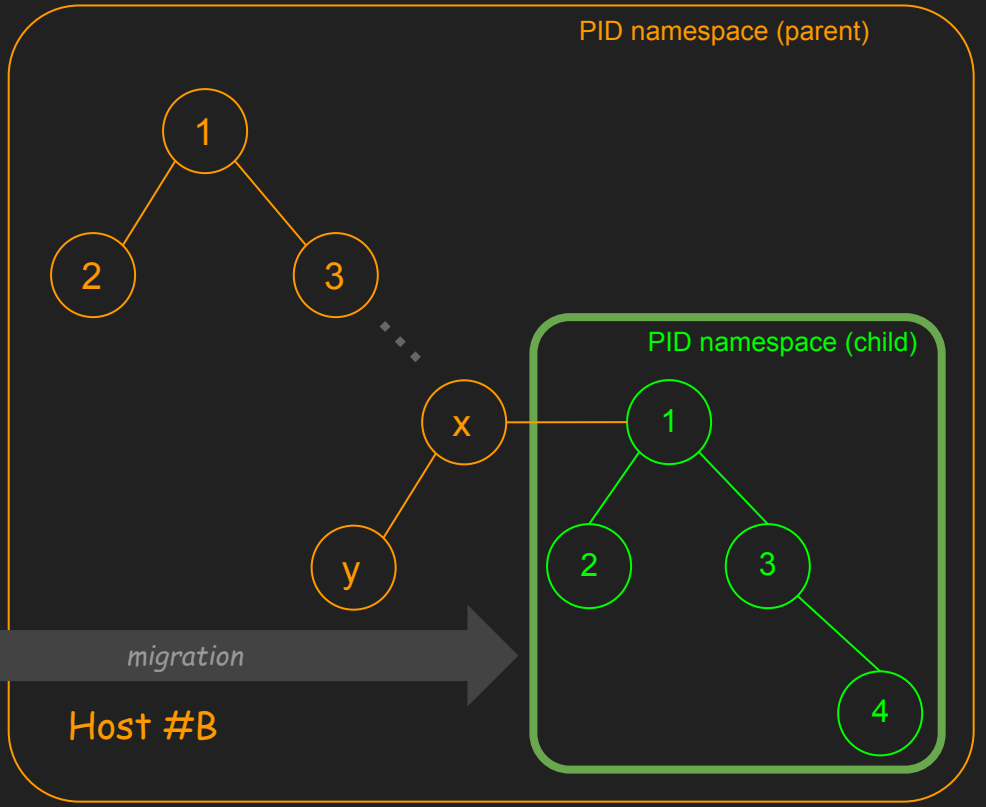
pid namespace

isolate the process ID number space

- enable container migration
  (isolated pid number space)

Host #A --(migration)--> Host #B



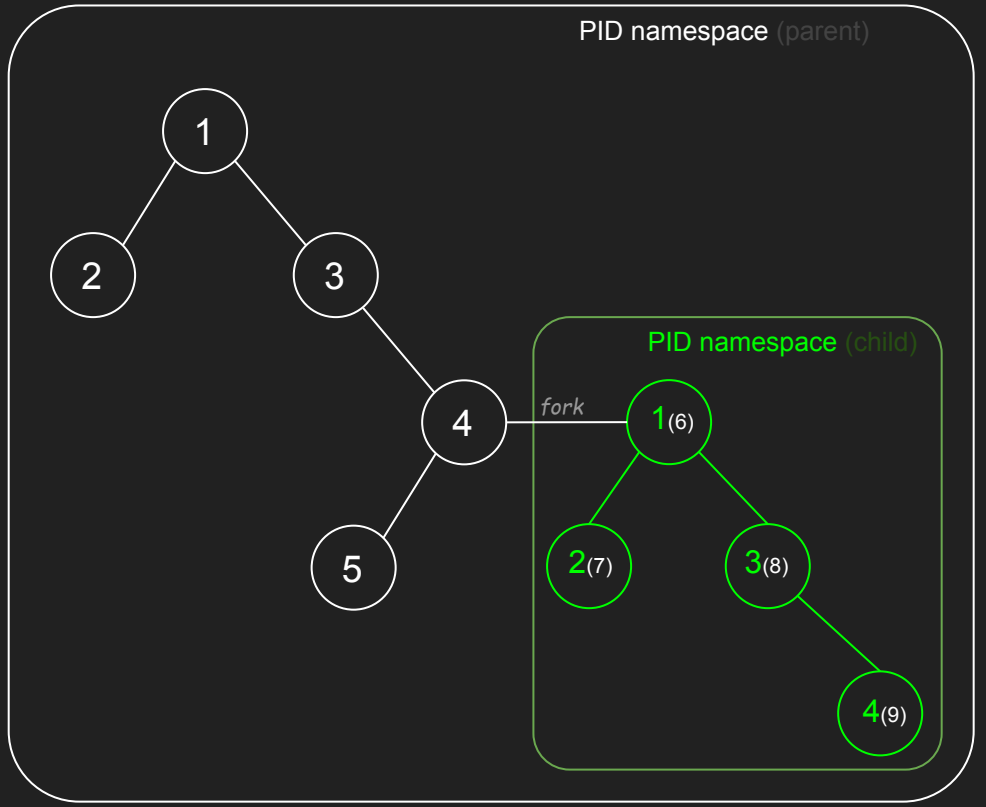


참고: 컨테이너 라이브 마이그레이션

pid namespace

isolate the process ID number space

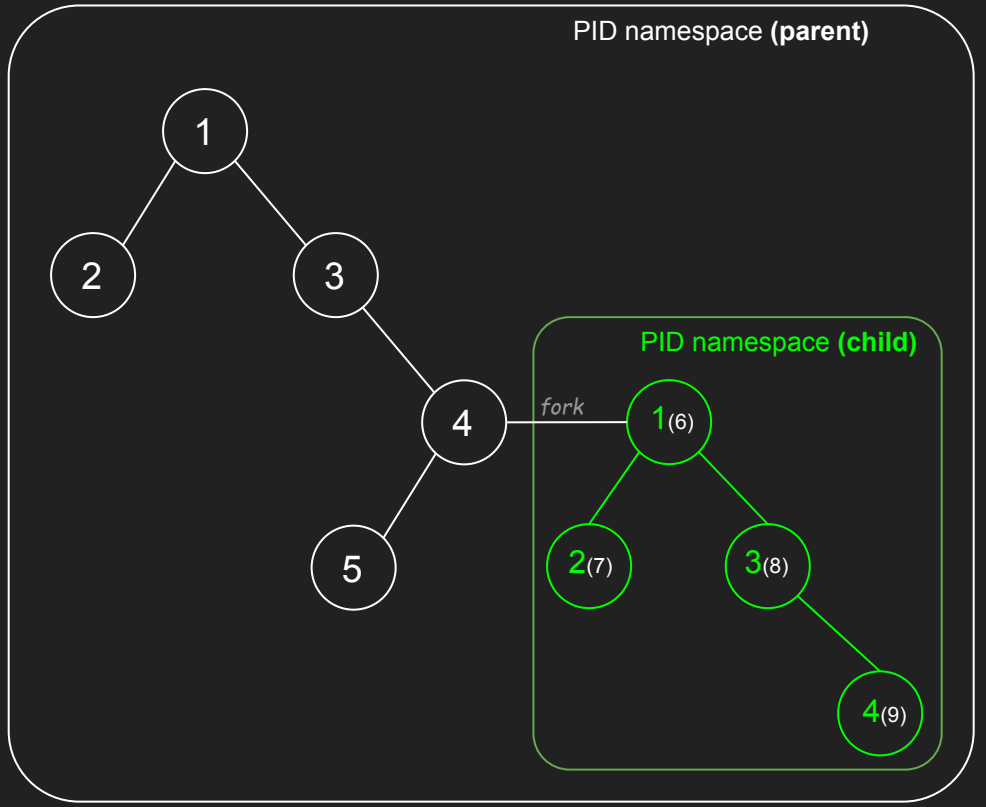
- unshare & fork (required)
- (forked) child becomes "1"
- child has "2 pids"



pid namespace

isolate the process ID number space

- nested hierarchical structure
  (parent & child namespace)
- parent sees processes in child



pid 1 in pid namespace

- signal handling
- reaping zombie, orphan
- lifecycle ~ pid1 dies, pid ns dies

(실습) pid namespace 생성

pid namespace 를 생성해 봅시다

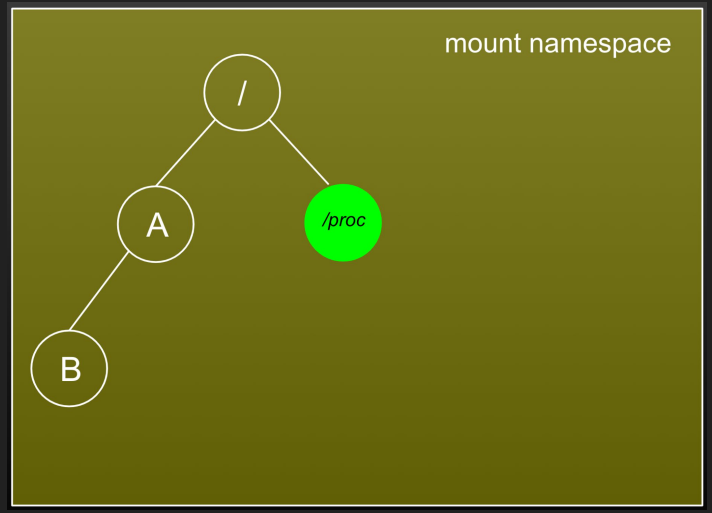
터미널 #1

```
# unshare -fp --mount-proc
```

unshare

- p : unshare the PID namespace
- f : fork
- --mount-proc : mount the proc filesystem

(실습) pid namespace 생성



--mount-proc : mount the proc filesystem

--mount-proc ? proc 파일시스템 마운트 옵션

터미널 #2

```
# ps -ef |  grep unshare

# lsns -p <pid>

# lsns <mnt inode>
```

```
  PID  PPID USER COMMAND
16921 13878 root unshare -fp --mount-proc
16922 16921 root └─-bash
```

(실습) pid namespace 생성

proc 파일시스템이란? (man page)

```
NAME
       proc - process information pseudo-filesystem

DESCRIPTION
       The  proc  filesystem  is a pseudo-filesystem which provides an
interface to kernel data structures.  It is commonly mounted at /proc.
```

proc 파일시스템이란?

/proc (procfs)

- pseudo filesystem (memory based virtual filesystem)
- 커널이 관리하는 시스템 정보의 제공과 제어
- 커널 데이터 구조의 접근을 쉽게 할 수 있음
- 시스템 모니터링과 분석에 활용

컨테이너 (pid namespace) 안에서 프로세스 정보를 조회하고 제어하기 위함

(실습) pid namespace 생성

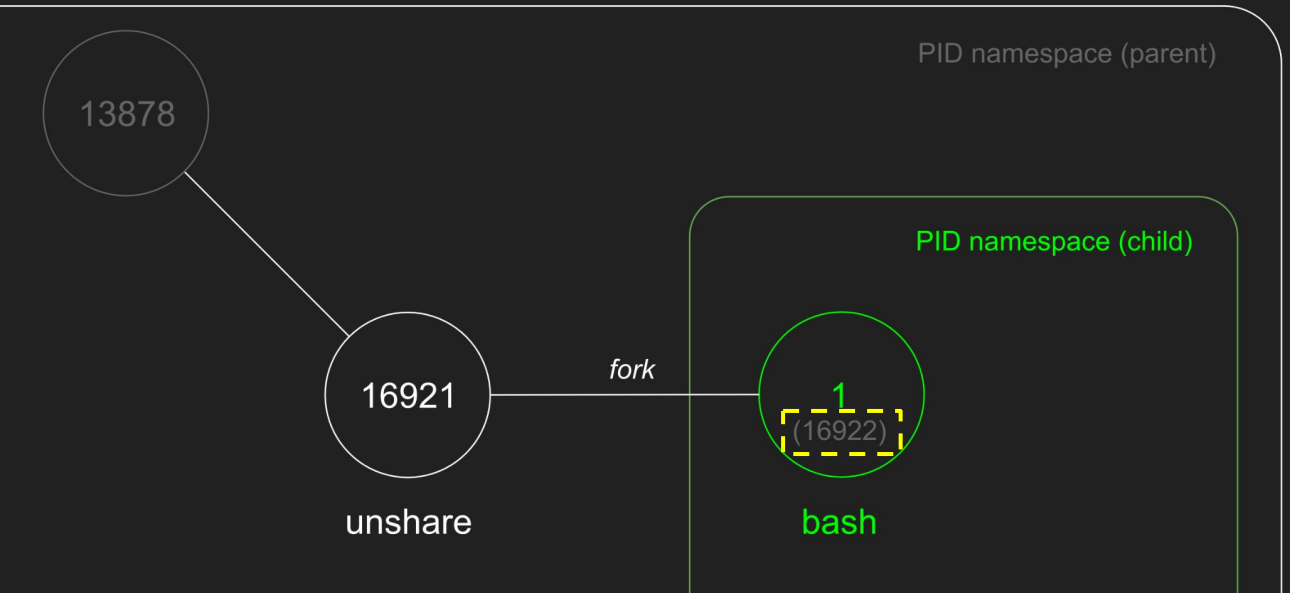
터미널 #1 (child namespace)

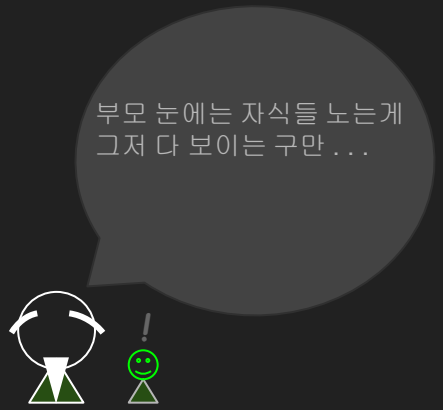
```
/# lsns -t pid -p 1
```

pid namespace 정보 확인

터미널 #2 (host ~ parent namespace)

```
# lsns -t pid -p 16922
```





(실습) pid namespace 생성

exit 하여 pid namespace를 닫아 주세요

터미널 #1

```
/# exit
```

(실습) **pid1** ~ signal handling

특별한 pid1의 시그널 처리를 확인해 봅시다

터미널 #1

```
# vi signal.py

import sys
import signal
import time

def signal_handler(signum, frame):
    print("Gracefully shutting down after receiving signal(%s) " % signum)
    # sys.exit(0)


if __name__ == "__main__":
    signal.signal(signal.SIGTERM, signal_handler)
    signal.signal(signal.SIGINT, signal_handler)
    while True:
        time.sleep(0.5)  # simulate work
          print("Interrupt me")
```

0.5초 sleep 하고 “Interrupt me”를 출력하는 무한루프 코드입니다.

SIGTERM과 SIGINT 시그널 발생 시 signal_handler를 호출합니다.

signal_handler는 signum을 포함한 메시지를 출력합니다.

종료코드(sys.exit)는 주석처리해 둡니다.

(실습) pid1 ~ signal handling

child pid namespace에서 signal.py를 실행

터미널 #1 (child namespace)

```
# unshare -fp python signal.py
Interrupt me
Interrupt me
Interrupt me
. . .
```

(실습) pid1 ~ signal handling

child pid namespace에서 signal.py를 실행

터미널 #1 (child namespace)

```
Interrupt me
Interrupt me
Interrupt me
. . .
```

터미널 #2 (host)

```
# ps -ef | grep signal.py
. . .
root     7617  7602  unshare -fp python signal.py
root     7618  7617  python signal.py
. . .
```

*호스트 상에서 **PID (7618)** 확인*

** pid는 각 자 실습환경마다 다릅니다.*

(실습) pid1 ~ signal handling

child namespace에 attach 해봅시다

터미널 #1 (child namespace)

```
. . .
Interrupt me
Interrupt me
Interrupt me
. . .
```

터미널 #2

```
# nsenter -m -p -t 7618

/#
```

pid namespace에 nsenter하여
pid1(signal.py) 확인

(실습) pid1 ~ signal handling

특별한 pid1은 signal.py

터미널 #1 (child namespace)

```
. . .
Interrupt me
Interrupt me
Interrupt me
. . .
```

터미널 #2 (child namespace)

```
/# ps -ef
root      1    0  python signal.py
root      2    0  -bash
root      6    2  ps -ef

/# exit
```

pid namespace에 nsenter하여
pid1(signal.py) 확인

(실습) pid1 ~ signal handling

시그널을 날려보세요

터미널 #1 (child namespace)

```
. . .
Interrupt me
Interrupt me
Interrupt me
. . .
```

터미널 #2 (host ~ parent namespace)

```
# kill -SIGHUP 7618
```

SYNOPSYS : kill -<signum> <pid>

* SIGINT, SIGTERM이외의 시그널은 무시됩니다 (단, SIGKILL 예외)

(실습) pid1 ~ signal handling

SIGINT와 SIGTERM 시그널도 보내보세요

터미널 #1 (child namespace)

```
. . .
Gracefully shutting down after receiving signal(2)
. . .
Gracefully shutting down after receiving signal(15)
. . .
```

터미널 #2 (host ~ parent namespace)

```
# kill -SIGINT 7618

# kill -SIGTERM 7618
```

코드에 작성한 대로 signal_handler에서 처리한 로그를 출력해 줍니다.

(실습) pid1 ~ signal handling

only SIGKILL로만 죽일 수 있음

터미널 #1 (child namespace)

```
. . .
Killed

#
```

child namespace에서 **pid1**인 signal.py가 죽으면서 해당 pid namespace는 닫힙니다.

터미널 #2 (host ~ parent namespace)

```
# kill -SIGKILL 7618
```

signal.py는 pid 1(init process) 이지만 시그널 처리를 안함
→ 사용자 구현에 의존

(실습) docker container ~ signal handling

도커 컨테이너의 pid1은 어떨까요? (시그널 처리를 할까요?)

터미널 #1 (child namespace)

```
# docker run --rm --name busybox busybox sleep 3600
```

터미널 #2 (host ~ parent namespace)

```
# ps aux | grep sleep
. . .
root      7941  docker run busybox sleep 3600
root      8017  sleep 3600
. . .
```

(실습) docker container ~ signal handling

pid1은 sleep 3600

터미널 #1 (child namespace)

```
# docker run --rm --name busybox busybox sleep 3600
```

터미널 #2 (host ~ parent namespace)

```
# docker exec busybox ps -ef
. . .
1   root    sleep 3600
. . .
```

(실습) docker container ~ signal handling

SIGKILL로만 죽일 수 있음

터미널 #1 (child namespace)

```
# docker run --rm --name busybox busybox sleep 3600
```

터미널 #2 (host ~ parent namespace)

```
# kill -SIGHUP 8017

# kill -SIGINT 8017

# kill -SIGTERM 8017

# kill -SIGKILL 8017
```

도커 역시 pid 1을 ...

→ 사용자 구현에 의존

(실습) docker ~ Simple init process

도커 컨테이너 기동 옵션 **--init**

터미널 #1 (child namespace)

```
# docker run --rm --name busybox --init busybox sleep 3600
```

도커는 **옵션**으로

→ 심플 init process를 제공합니다

(실습) docker ~ Simple init process

아까와는 다르게 docker-init 이라는 프로세스가 보입니다

터미널 #1 (child namespace)

```
# docker run --rm --name busybox --init busybox sleep 3600
```

터미널 #2 (host ~ parent namespace)

```
# ps aux | grep sleep
. . .
root      8399  7687  docker run --rm --name busybox
--init busybox sleep 3600
root      8487  8458  /sbin/docker-init -- sleep 3600
root      8537  8487  sleep 3600
. . .
```

(실습) docker ~ Simple init process

컨테이너 안에서 보면 ~ pid1은 docker-init

터미널 #1 (child namespace)

```
# docker run --rm --name busybox --init busybox sleep 3600
```

터미널 #2 (host ~ parent namespace)

```
# docker exec busybox ps -ef
. . .
1   root    /sbin/docker-init -- sleep 3600
8   root     sleep 3600
```

(실습) docker ~ Simple init process

docker-init (pid1)이 시그널 처리를 해줌

터미널 #1 (child namespace)

```
/# docker run --rm --name busybox --init busybox sleep 3600

#
```

터미널 #2 (host ~ parent namespace)

```
# kill -SIGHUP 8487

--- 오메 ~ 바로 죽어 버려야 ---

# kill -SIGINT 8487

# kill -SIGTERM 8487

# kill -SIGKILL 8487
```

수고 많으셨습니다 다음편에서 봬요 :-)

목차 보기