Containers – A look under the hood

Gerlof Langeveld

AT COMPUTING
ATYPICAL OPEN SOURCE GURUS

Download: `https://www.atoptool.nl/oss/cddkit.tgz`

Conventional UNIX approach

Conventional approach: all processes run in one eco system

- unambiguous notion of
  - hostname
  - PID numbers
  - mounted filesystems
  - network stack (open ports, interfaces)
  - IPC objects (shared memory, message queues, semaphores)
  - users (names, uid's)

- not necessarily uniform view on filesystem
  - each process might have different root directory (**`chroot`**: 1979)

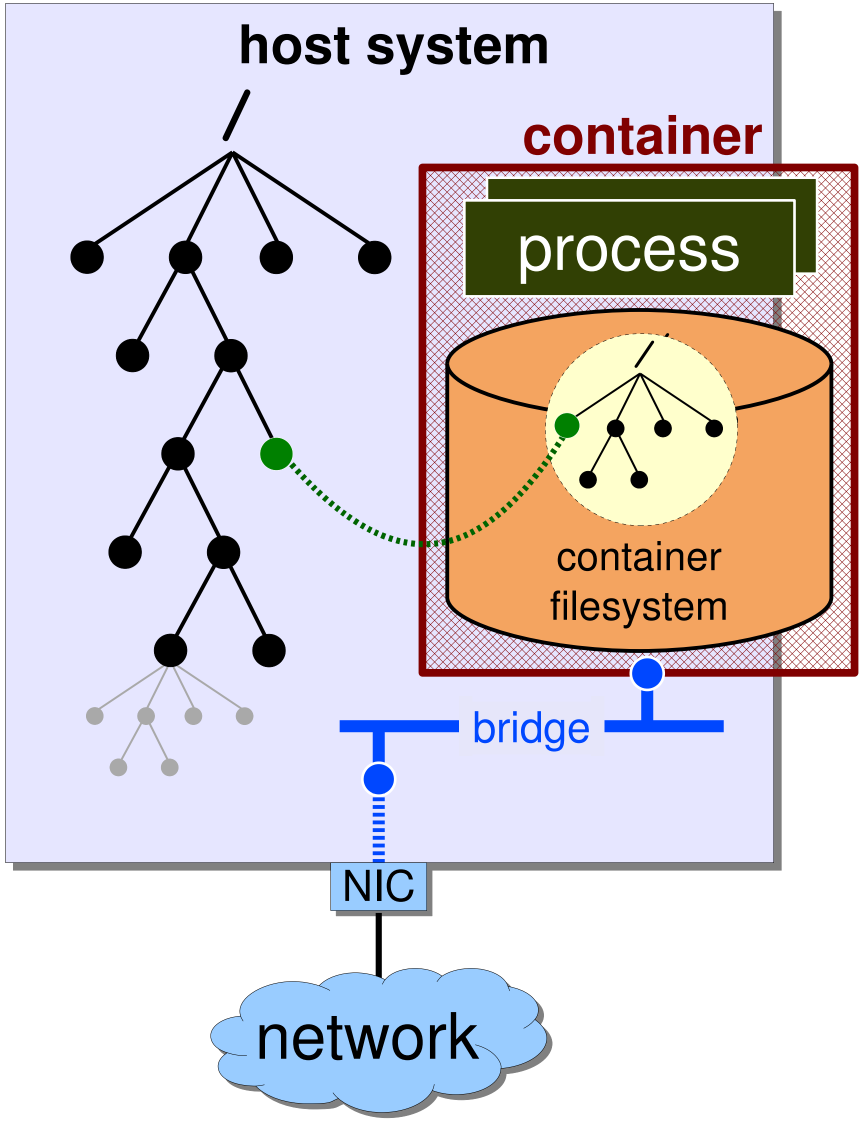
- with root identity: *all* privileged actions allowed
  with non-root identity: *no* privileged actions allowed

- hardly possible to control resource consumption

Containerized approach (1)

Process(es) in container should be

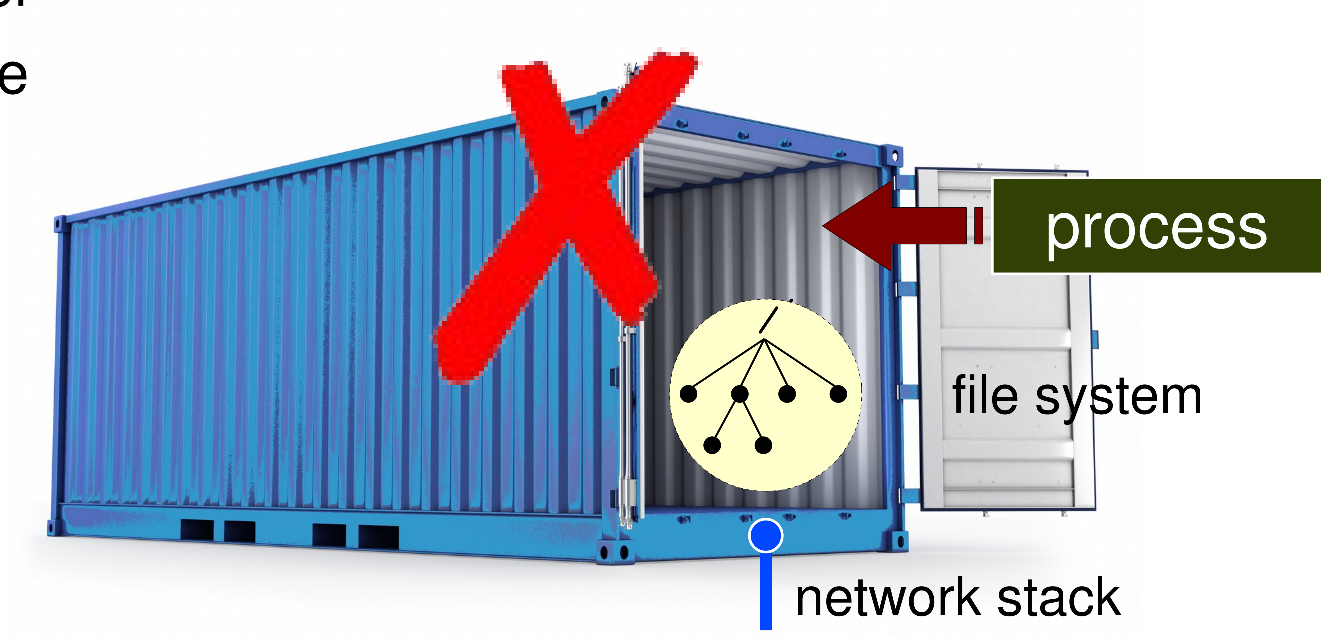
- isolated from other processes on host
  - with private filesystem
    including private mounted storage
  - with private hostname
  - with private PID numbering
  - with private network stack
  - with limited privileges
    (even when running under root identity)
  - with limited or guaranteed utilization of
    hardware resources (CPU, memory, ...)



Containerized approach (2)

Containerized application process – **unrealistic view**

- prepare and furnish container
  - mini filesystem from image
  - private network
  - PID number generator
  - ....



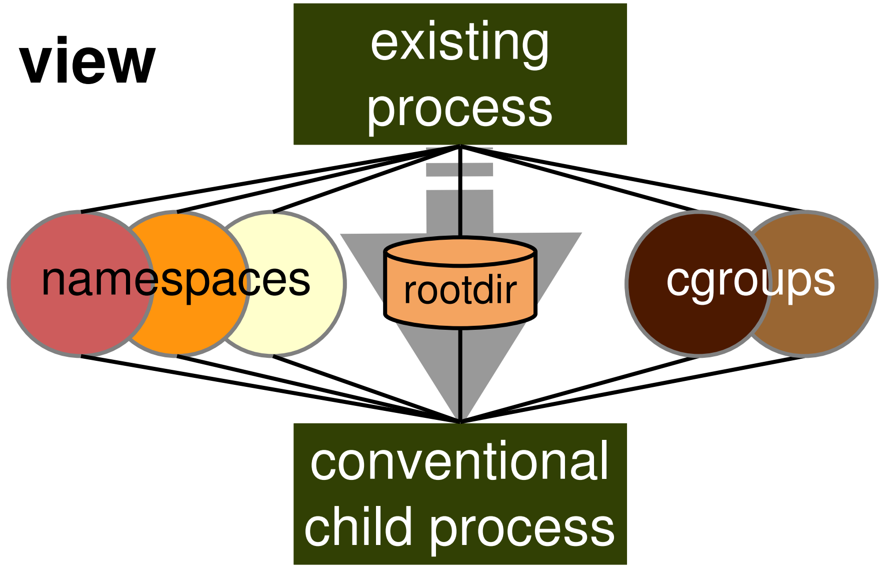
and finally...

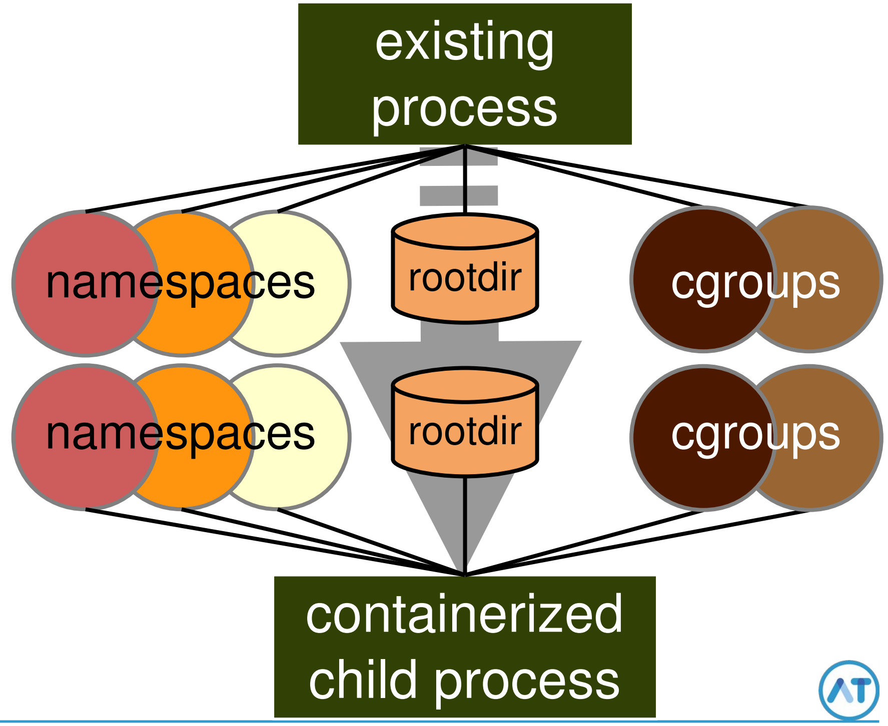
- place new process in container

Containerized approach (3)

Containerized application process – **realistic view**

- administration of process/thread (task) in kernel refers to
  - root directory
  - namespaces to share environments, like network stack or PID numbering
  - control groups (cgroups) to limit/guarantee resource utilization
- conventional process
  - shares namespaces and control groups
- *containerized process is still native process*
  - however, it (partly) gets its own namespaces and control groups





Container implementation

Linux kernel mechanisms to implement container

- isolated hostname – *namespace ‘uts’*
- isolated IPC objects – *namespace ‘ipc’*
- isolated PID numbers – *namespace ‘pid’*
- isolated network stack – *namespace ‘net’*
- isolated mount points – *namespace ‘mnt’*
- isolated users – *namespace ‘user’*

- private filesystem – *chroot/pivot_root*
- distinct privileges – *capabilities*

- limited/guaranteed CPU utilization – *cgroup ‘cpu’ & ‘cpuset’*
- limited/guaranteed memory utilization – *cgroup ‘memory’*
- limited/guaranteed disk utilization – *cgroup ‘blkio’ / ‘io’*

Goal: *create containerized process without using known implementation*

Process-related details

Pseudo-filesystem **proc**

- usually mounted as **/proc**
  - access to system-level kernel data

```
$ cat /proc/stat
cpu  59118 12782 71090 166419102 1916 6832 6161 44789 0 0
cpu0 27871 7813 44158 63085572 947 3051 3443 21528 0 0
....
```

  - access to process-level kernel data

```
$ ps
    PID TTY          TIME CMD
  67022 pts/0    00:00:00 bash
  67069 pts/0    00:00:00 ps

$ ls -l /proc/67022
....
-r--r--r--. 1 gerlof gerlof 0 Sep  7 09:54  cgroup
dr-x--x--x. 2 gerlof gerlof 0 Sep  7 09:54  ns
lrwxrwxrwx. 1 gerlof gerlof 0 Sep  7 09:54  root -> /
-r--r--r--. 1 gerlof gerlof 0 Sep  7 09:54  status
```

e.g. capabilities

Namespaces

Namespace – general

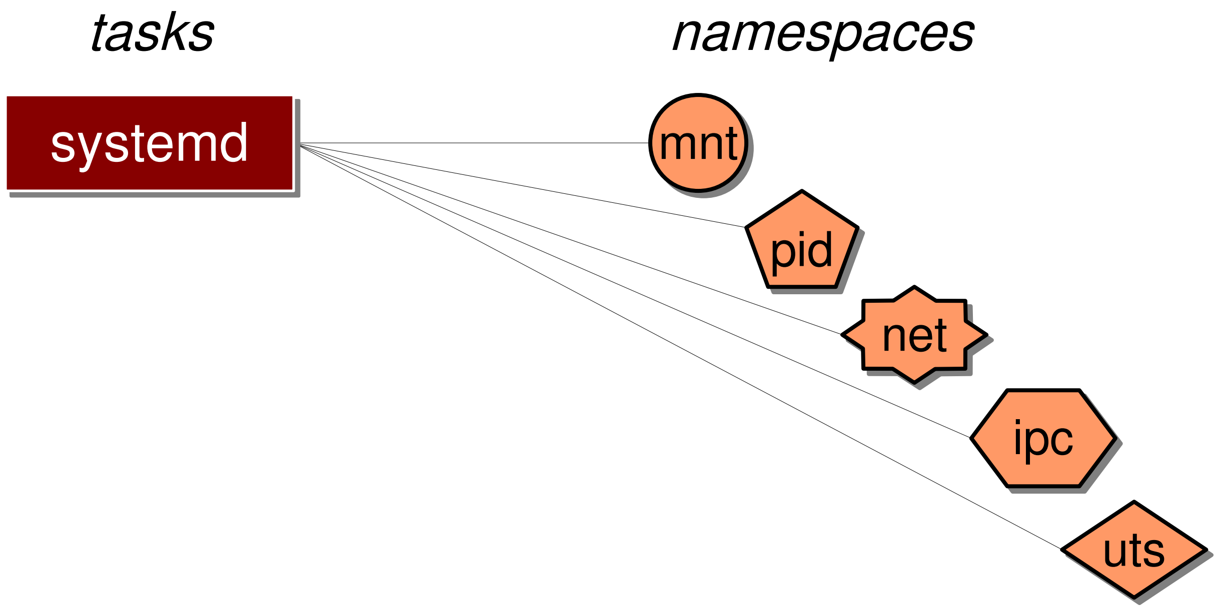
Namespace

- allows processes to share dedicated collection of resources
  - hostnames
  - process IDs
  - network interfaces & ports
  - mounted filesystems
  - ....
- every type of namespace has own behavior
- creation requires **`CAP_SYS_ADMIN`** capability, except for user namespace
- allows separation of (groups of) processes without allocating VMs

Introduction namespaces (1)

Processes share namespaces

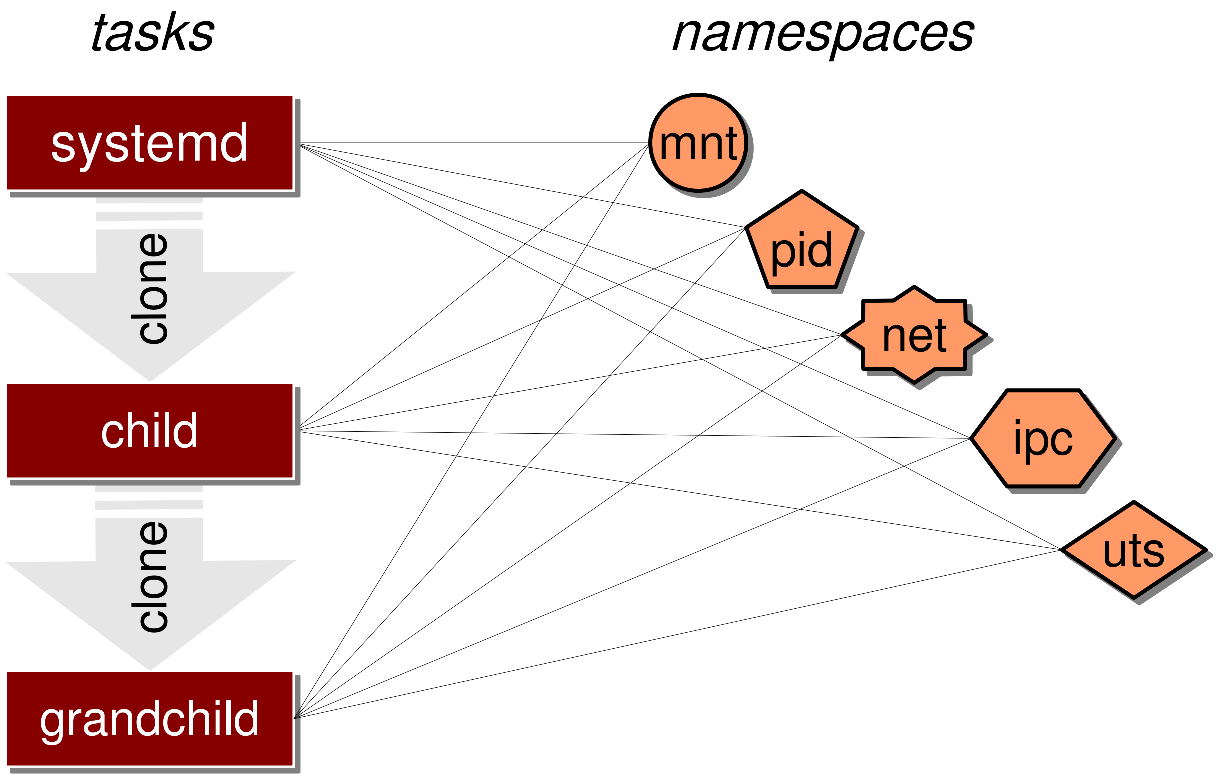
- process (task) administration in kernel refers to *namespaces*
  - describe environments, like hostname, network stack or pid numbering



Introduction namespaces (2)

Processes share namespaces

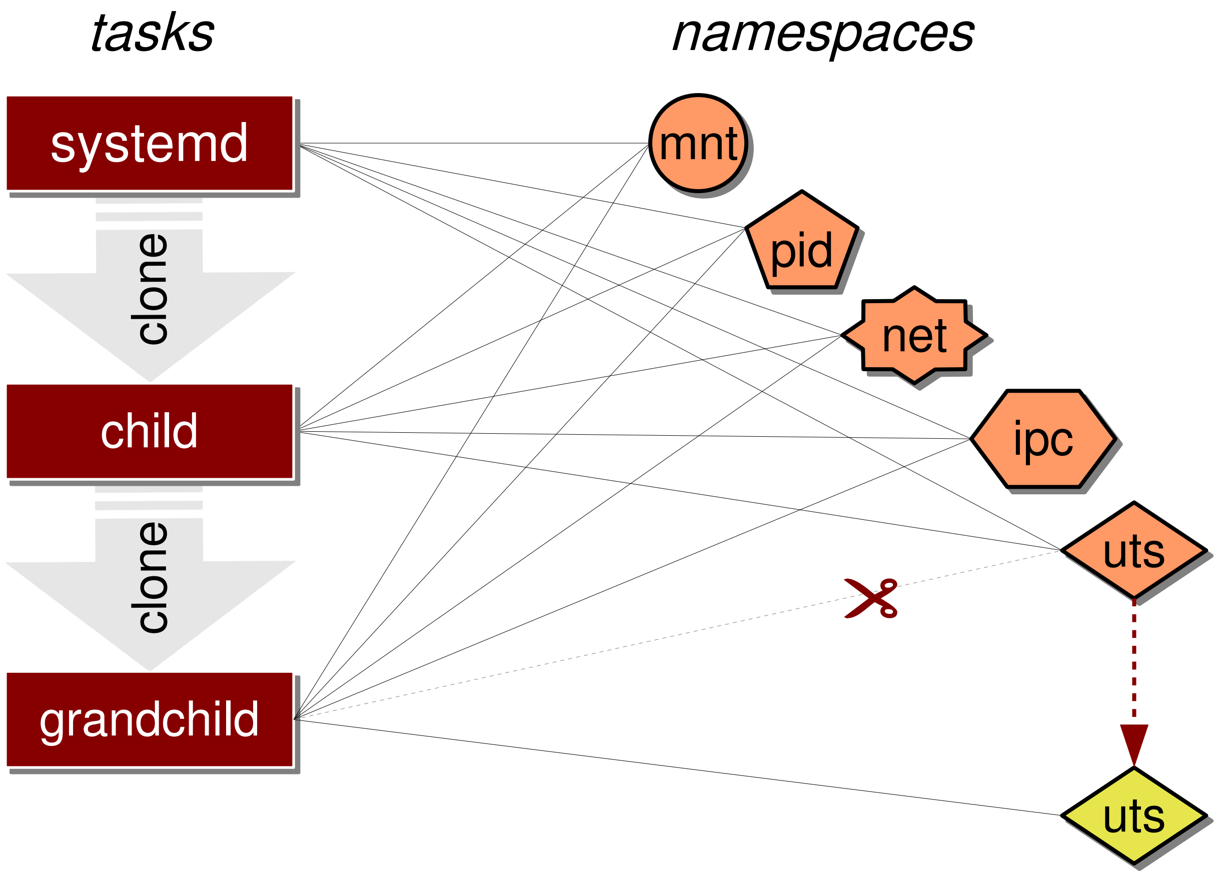
- process (task) administration in kernel refers to *namespaces*
  - describe environments, like hostname, network stack or pid numbering
- child processes inherit reference to namespaces
  - share same environments



Introduction namespaces (3)

Processes share namespaces

- process (task) administration in kernel refers to *namespaces*
  - describe environments, like hostname, network stack or pid numbering
- child processes inherit reference to namespaces
  - share same environments
- process can unshare namespace
  - gets private namespace
    - empty, or
    - modifyable copy



Namespaces – reference

Namespace sharing

- every process refers to namespaces

inode#

```
$ ls -l /proc/$$/ns
....
lrwxrwxrwx 1 gerlof gerlof 0 Feb 27 08:40  ipc -> ipc:[4026531839]
lrwxrwxrwx 1 gerlof gerlof 0 Feb 27 08:40  mnt -> mnt:[4026531840]
lrwxrwxrwx 1 gerlof gerlof 0 Feb 27 08:40  net -> net:[4026531968]
lrwxrwxrwx 1 gerlof gerlof 0 Feb 27 08:40  pid -> pid:[4026531836]
....
lrwxrwxrwx 1 gerlof gerlof 0 Feb 27 08:40  uts -> uts:[4026531838]
```

- same inode (value between `[....]`) means same namespace

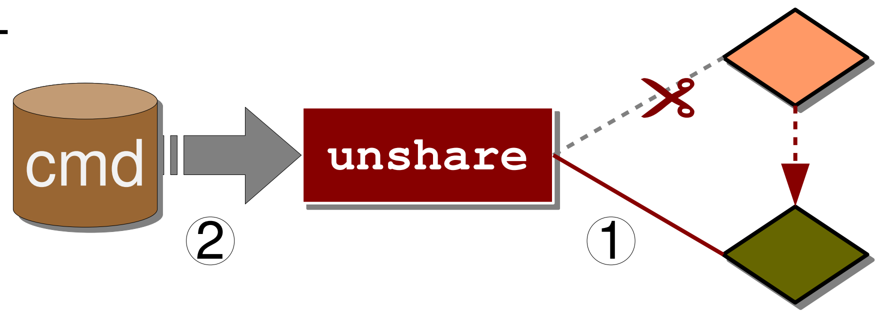
```
$ ls -l /proc/$$/ns/uts
lrwxrwxrwx 1 gerlof gerlof 0 Feb 27 08:40  uts -> uts:[4026531838]

# ls -l /proc/1/ns/uts
lrwxrwxrwx 1 root   root   0 Feb 27 08:40  uts -> uts:[4026531838]
```

  - child process inherits association with namespace from parent
  - namespace vanishes when all processes disconnected, except for *persistent* namespace

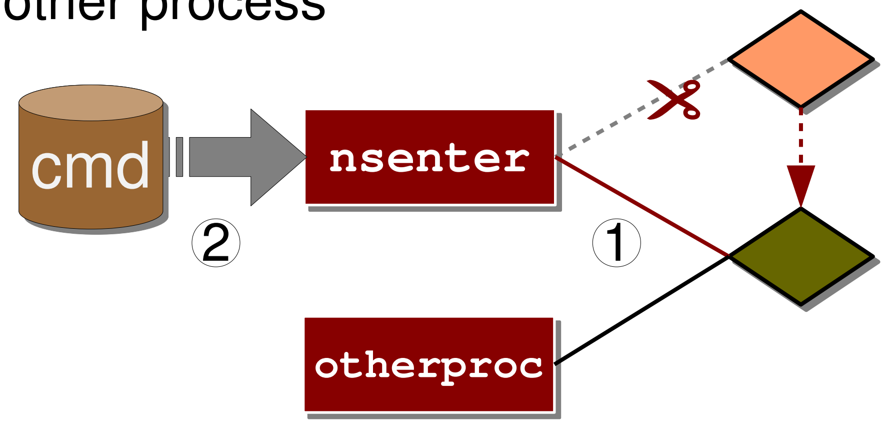
Namespaces – commands

Namespace-related commands

- command ***unshare*** *[-umin...]* ***cmd...***
  - unshare namespace(s) for process
  - execute required command in process
  - using system call **unshare(2)**



- command ***nsenter*** *[-umin...]* ***-t pid cmd...***
  - connect process with existing namespace(s) of other process
  - execute required program in process
  - using system call **setns(2)**



Other namespace-related system calls

- create new process with new namespace(s)
  - system call **clone(2)**

Namespaces – uts

UTS namespace – hostname isolation

- example *uts* namespace (flag **-u**)

```
$ ls -l /proc/$$/ns/uts
lrwxrwxrwx 1 gerlof gerlof 0 Feb 27 08:40  uts -> uts:[4026531838]

$ hostname
myhost

$ sudo unshare -u bash
[root@myhost]# ls -l /proc/$$/ns/uts
lrwxrwxrwx 1 root   root   0 Feb 27 08:41  uts -> uts:[4026532505]

[root@myhost]# hostname otherhost

[root@myhost]# bash
[root@otherhost]# exit

[root@myhost]# exit

$ hostname
myhost
```

Build container – collection of shell scripts

step1:

```
unshare -u  bash step2
```

step2:

```
hostname mycontainer
bash
```

Make **step1** executable (once):

```
$ chmod +x step1
```

Run **step1**:

```
$ sudo ./step1
```

Namespaces – ipc

IPC namespace – isolation of IPC objects

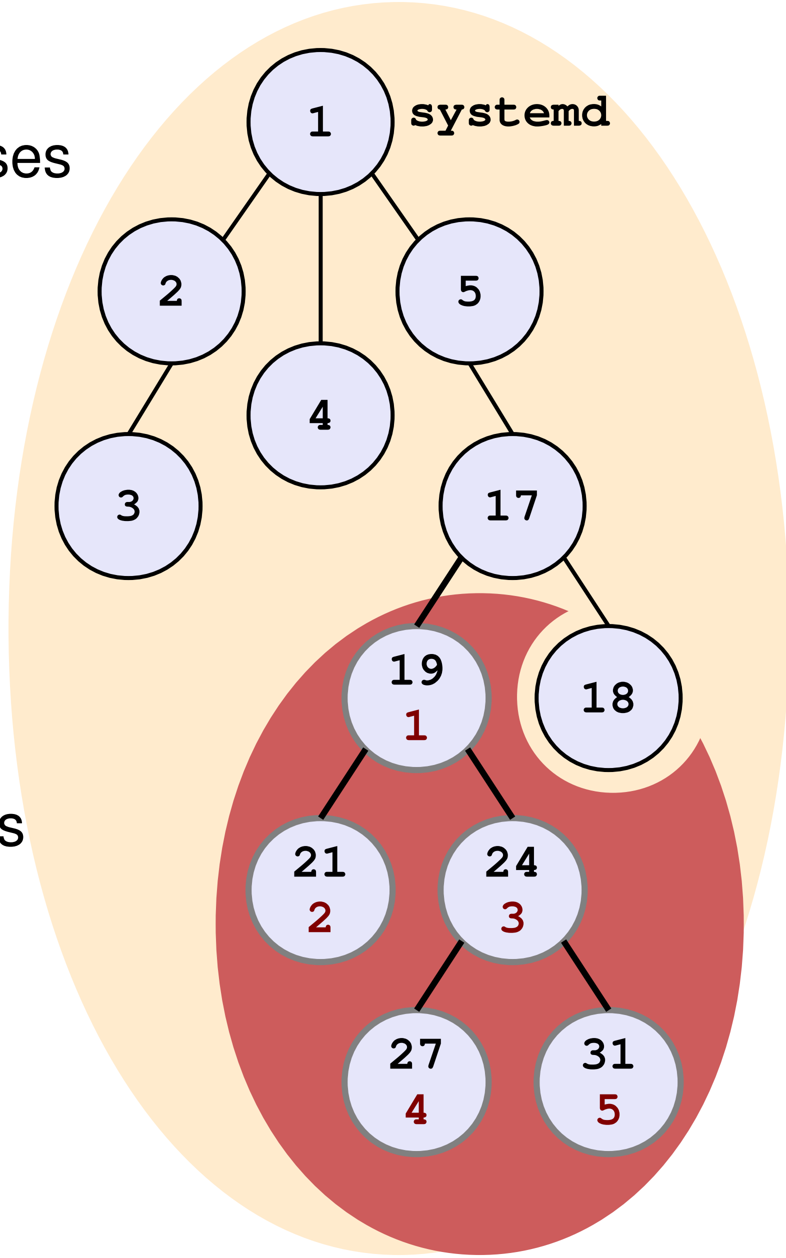
- example *ipc* namespace (flag **-i**)

```
$ ipcs
------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
0xfeedbabe 0          gerlof     666        0            0

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x00000000 327680     gerlof     600        339968     2          dest
0x00000000 327681     gerlof     600        339968     2          dest

------ Semaphore Arrays --------
key        semid      owner      perms      nsems
0x00178749 9          root       600        2

$ sudo unshare -i bash
# ipcs                 ☞ no ipc objects!
# ipcmk -M 4096
# ipcs
------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0xdb72639f 0          root       644        4096       0
```

Namespaces – pid (1)

PID namespace – PID isolation

- defines PID numbering scheme for set of processes
  - remount of `/proc` needed
- nested namespace
  - when process (17) unshares pid namespace, its child process gets
    - next PID (19) in ancestor namespace(s)
    - PID 1 in new namespace
  - when child process spawns grandchild process
    - next PID in ancestor namespace(s)
    - PID 2 in new namespace
- process with PID 1 in any namespace
  - reaper for orphaned children in namespace
  - when it terminates, all processes in namespace terminate



Namespaces – pid (2)

PID namespace – PID isolation

- example *pid* namespace (flag **-p**)

```
$ sudo unshare -p  --fork  --mount-proc  bash
# sleep 300&

# ps -ef
UID       PID  PPID  C STIME TTY          TIME CMD
root        1     0  0 11:53 pts/12   00:00:00 bash
root       37     1  0 11:54 pts/12   00:00:00 sleep 300
root       38     1  0 11:54 pts/12   00:00:00 ps -ef

# ls -l /proc
total 0
dr-xr-xr-x  9 root root  0 Feb 27 11:53  1
dr-xr-xr-x  9 root root  0 Feb 27 11:54  37
dr-xr-xr-x  9 root root  0 Feb 27 11:54  39
....
```

- on host system

```
$ ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root      9728  9727  0 11:54 pts/12   00:00:00 bash
root      9782  9728  0 11:54 pts/12   00:00:00 sleep 300
```

Build container – collection of shell scripts

step1:
```
unshare -u  bash step2
```

step2:
```
hostname mycontainer
unshare -p --fork --mount-proc  bash step3
```

step3:
```
bash
```

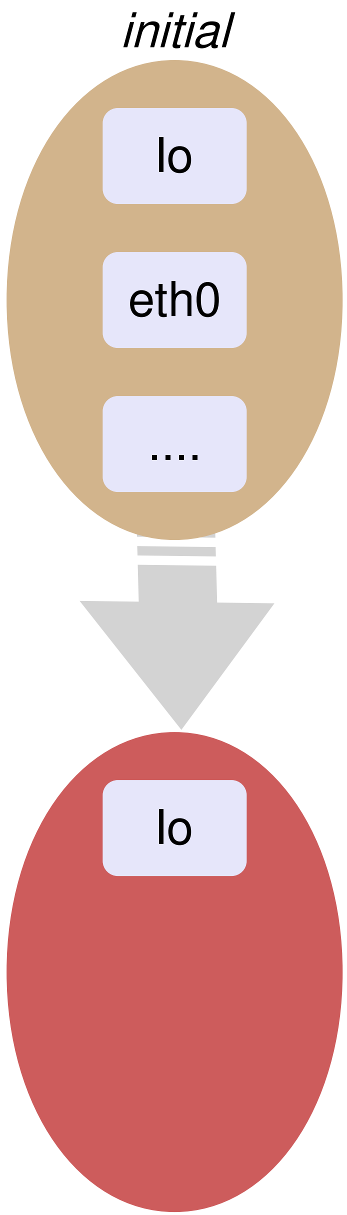
Run **step1**:
```
$ sudo ./step1
```

Namespaces – net (1)

NET namespace – network isolation

- defines network stack with own ports, routes and network interfaces
- process might create new network namespace
  - initially only loopback interface (state down)
  - new interfaces can be added
    - assign IP address
    - define routing
- physical interfaces can only be assigned to one namespace
- namespaces can be interconnected via *veth* pairs
- some kernel parameters are namespaced

```
# echo 512 > /proc/sys/net/ipv4/ip_unprivileged_port_start
```



Namespaces – net (2)

NET namespace – network isolation

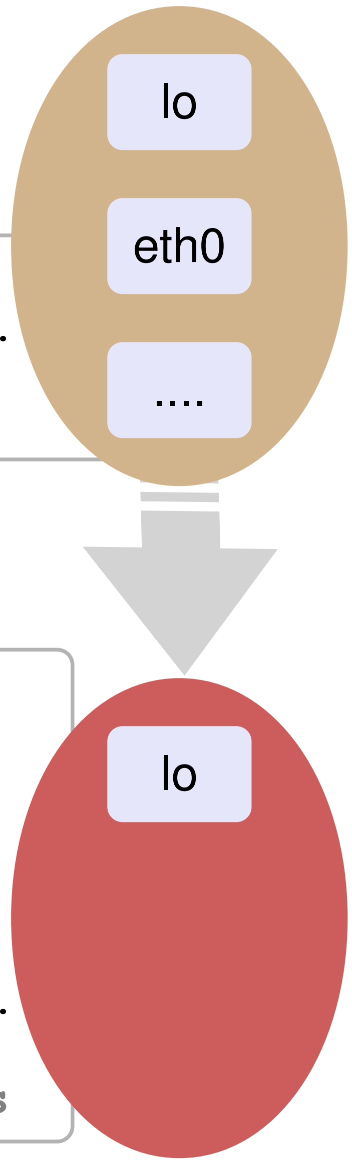
- example *network* namespace (flag **–n**)
  - shell using *initial* netns:

```
$ ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN ...
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc ... state UP
....
```

  - start shell using *new* netns:

```
$ sudo unshare –n  bash
# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noop state DOWN ....

# ip link set dev lo up
# ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN ...

# ss –ta                        # no LISTEN or ESTABLISHED TCP ports
```

initial



Namespaces – net (3)

NET namespace – setup virtual bridge between two namespaces

- shell using *initial* netns: determine PID of shell assigned to *new* netns

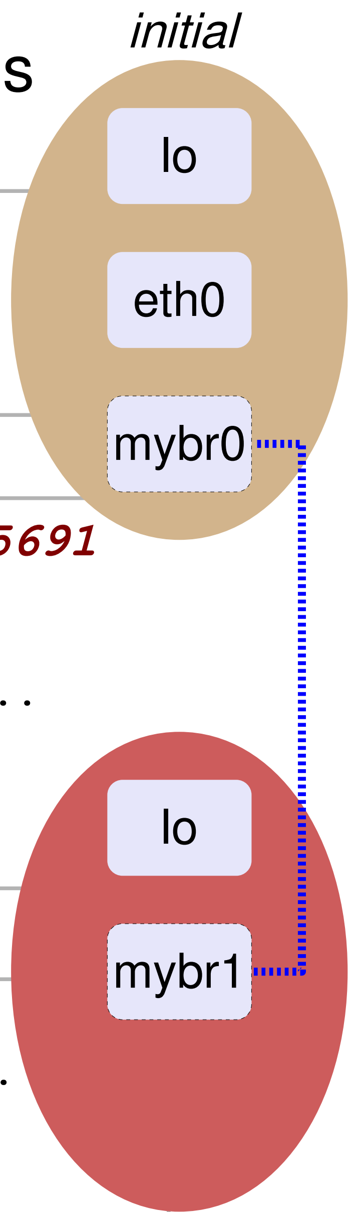
```
# ps -f
UID          PID  PPID  C STIME TTY          TIME CMD
root       25690 29335  0 16:18 pts/11   00:00:00 sudo unshare -n bash
root       25691 25690  0 16:18 pts/11   00:00:00 bash
```

- shell using *initial* netns: create bridge and configure one part

```
# ip link add name mybr0 type veth peer name mybr1 netns 25691
# ip addr
....
15: mybr0@if2: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN ....

# ip addr add 192.168.47.11/24 dev mybr0
# ip link set dev mybr0 up
```

- shell using *new* netns: configure other part

```
# ip addr
2: mybr1@if15: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN ...

# ip addr add 192.168.47.12/24 dev mybr1
# ip link set dev mybr1 up
# ssh 192.168.47.11          # ssh login to ´host´ via bridge
```



Build container – collection of shell scripts

step1:
```
unshare -u  bash step2
```

step2:
```
hostname mycontainer
unshare -p --fork --mount-proc  bash step3
```

step3:
```
unshare -n  bash step4
```

step4:
```
bash
```

Run **step1**:

```
$ sudo ./step1
```

Build container – collection of shell scripts

step1:

```
unshare -u  bash step2
```

step2:

```
hostname mycontainer
unshare -p --fork --mount-proc  bash step3
```

step3:

```
unshare -n  bash step4
```

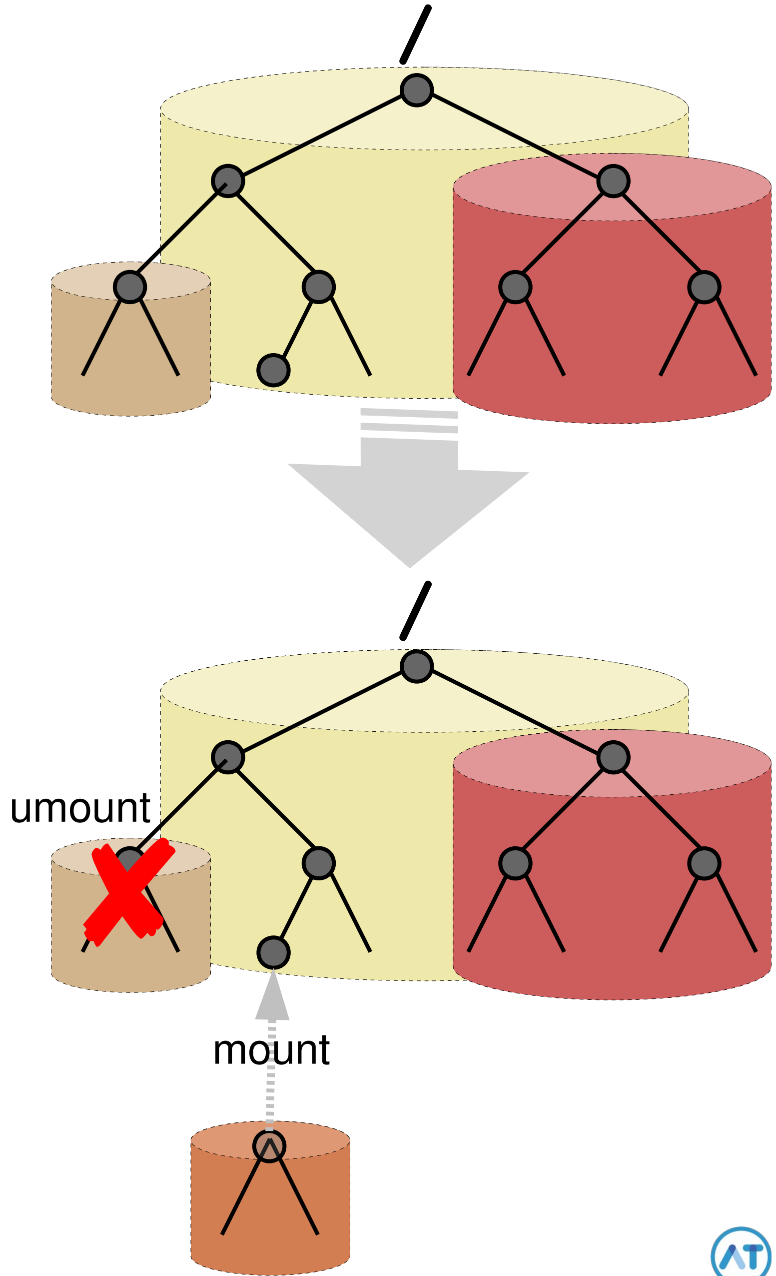
step4 (copy of **step4skel**):

```
ip link set dev lo up

nsenter -n -t 1 ip link add name mybr0 type veth peer name mybr1 netns $$
nsenter -n -t 1 ip addr add 192.168.47.11/24 dev mybr0
nsenter -n -t 1 ip link set dev mybr0 up

ip addr add 192.168.47.12/24 dev mybr1
ip link set dev mybr1 up

bash
```

Namespaces – mnt (1)

MNT namespace – mount point isolation

- mount points defined per mount namespace
- processes connected to same namespace share same mount points
- commands like **`mount`** and **`df`**
  - read 'file' **`/proc/self/mountinfo`** instead of file **`/etc/mtab`**
- when process started with new namespace
  - inherit mount points hierarchically
  - mount or unmount filesystems without affecting mount points of processes connected to other namespace(s)



Namespaces – mnt (2)

MNT namespace – mount point isolation

- example *mount* namespace (flag **-m**)

```
$ df
Filesystem     1K-blocks      Used  Available Use% Mounted on
/dev/sda2       26203648  14498208   11705440  56% /
/dev/sda1      249935632 215837452   34098180  87% /data
/dev/sda6       62885888  32212684   30673204  52% /home

$ sudo unshare -m  bash

# umount /data
# mount  /dev/sdb1 /mnt
# df
Filesystem     1K-blocks      Used  Available Use% Mounted on
/dev/sda2       26203648  14498208   11705440  56% /
/dev/sda6       62885888  32212684   30673204  52% /home
/dev/sdb1      209614848 163866884   45747964  79% /mnt
# exit

$ df
Filesystem     1K-blocks      Used  Available Use% Mounted on
/dev/sda2       26203648  14498208   11705440  56% /
/dev/sda1      249935632 215837452   34098180  87% /data
/dev/sda6       62885888  32212684   30673204  52% /home
```

Build container – collection of shell scripts

step1:

```
unshare -u  bash step2
```

step2:

```
hostname mycontainer
unshare -p --fork --mount-proc  bash step3
```

step3:

```
unshare -n  bash step4
```

step4:

```
ip link set dev lo up

nsenter -n -t 1 ip link add name mybr0 type veth peer name mybr1 netns $$
nsenter -n -t 1 ip addr add 192.168.47.11/24 dev mybr0
nsenter -n -t 1 ip link set dev mybr0 up

ip addr add 192.168.47.12/24 dev mybr1
ip link set dev mybr1 up

unshare -m  bash step5
```

step5:

```
bash
```

Namespaces – persistent

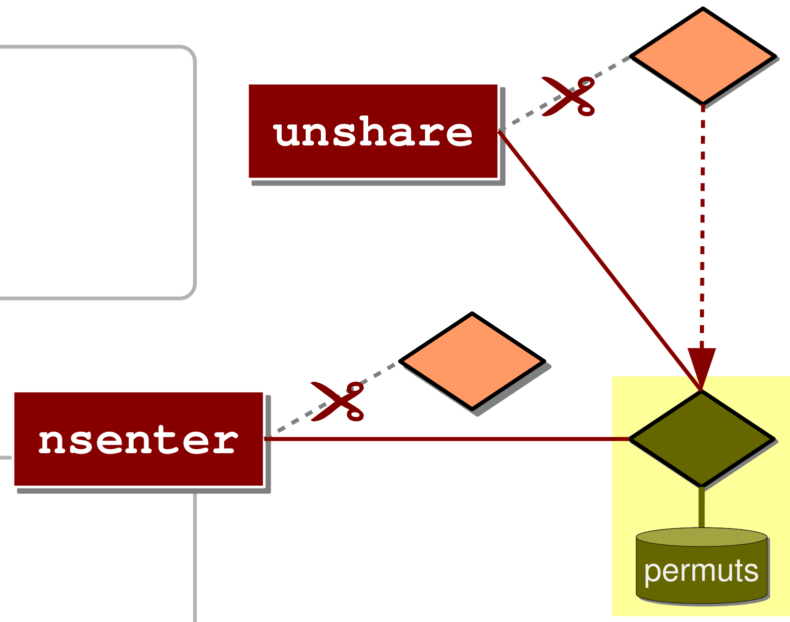
Persistent namespace

- preserve namespace by bind-mounting it to file
  - manually

```
$ sudo unshare --uts bash
# hostname myhost
# touch /tmp/permuts
# mount --bind /proc/self/ns/uts /tmp/permuts
# exit
```

  - or by command **unshare**

```
$ touch /tmp/permuts
$ sudo unshare --uts=/tmp/permuts bash
# hostname myhost
# exit
```



- connect persistent namespace to process

```
$ sudo nsenter --uts=/tmp/permuts bash
# hostname
myhost
```

Namespaces and Docker

Command **`docker run`**

- hostname isolation
  - share with host: **`--uts=host`**
  - share with other container: **`--uts=container:`*`CID`***
- PID isolation
  - share with host: **`--pid=host`**
  - share with other container: **`--pid=container:`*`CID`***
- network isolation
  - share with host: **`--network=host`**
  - share with other container: **`--network=container:`*`CID`***
- IPC isolation
  - share with host: **`--ipc=host`**
  - share with other container: **`--ipc=container:`*`CID`***
- namespaced kernel parameters
  - set specific parameter: **`--sysctl=net.ipv4.ip_unprivileged_port_start=512`**

example

Containers – A look under the hood

Modified
root directory

Modified root directory – introduction

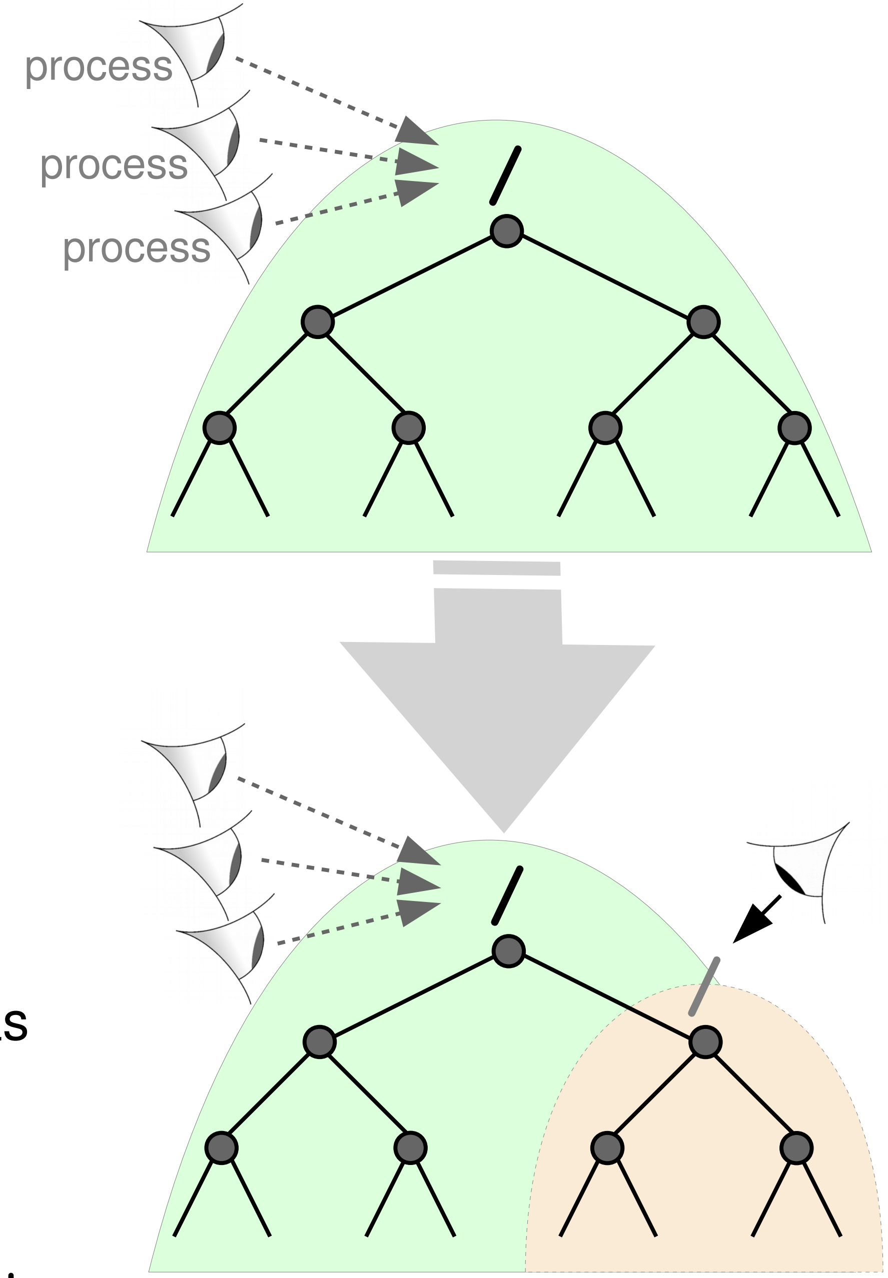
Every process has own root directory

- inherited from parent process

- usually all processes inherit root directory of entire filesystem from **`systemd`** (PID 1)

- for process with alternative root directory
  - prepare directory as root directory
    - executable file of application
    - shared libraries used by application
    - configuration files, data files, ...

  - activate new process with prepared directory as root directory using command **`chroot`**
    and/or
    change root directory for all processes in mount namespace with command **`pivot_root`**



Modified root directory – example (1)

Example: run **bash** (and **cat**) with limited private filesystem

- create directory **topdir** as root directory and create directories underneath

```
$ mkdir -p topdir/bin topdir/lib64 topdir/etc topdir/root
```

- copy/create some files into new directory tree

```
$ cp /bin/cat /bin/bash                         topdir/bin
$ echo root:x:0:0::/root:/bin/bash            > topdir/etc/passwd
$ echo 'PS1="[\u@\h \W]# "'                   > topdir/root/.bash_profile
```

- copy required shared libraries into new directory tree

```
$ ldd /bin/bash /bin/cat
/bin/bash:
   libtinfo.so.5 => /lib64/libtinfo.so.5 (0x00007f312b24f000)
   libdl.so.2 => /lib64/libdl.so.2 (0x00007f312b04b000)
   libc.so.6 => /lib64/libc.so.6 (0x00007f312ac7d000)
   /lib64/ld-linux-x86-64.so.2 (0x00007f312b479000)
/bin/cat:
   libc.so.6 => /lib64/libc.so.6 (0x00007f4a4679e000)
   /lib64/ld-linux-x86-64.so.2 (0x00007f4a46b6c000)

$ cp /lib64/libc.so.6  /lib64/ld-linux-x86-64.so.2  topdir/lib64
$ cp /lib64/libdl.so.2 /lib64/libtinfo.so.5         topdir/lib64
```

Modified root directory – example (2)

Example: run **bash** (and **cat**) with limited private filesystem – cont'd

- activate **bash** using private filesystem

```
$ sudo chroot topdir  bash --login
[root@myhost /]# ls -l
bash: ls: command not found

[root@myhost /]# echo /*
/bin  /etc  /lib64  /root

[root@myhost /]# echo /bin/*
/bin/cat

[root@myhost /]# cat /etc/passwd
root:x:0:0::/root:/bin/bash

[root@myhost /]# ps -f
bash: ps: command not found

[root@myhost /]# echo $$
30719

[root@myhost /]# exit
$
```

Alternatively use:

pivot_root newroot oldroot

must be mount point

Build container – collection of shell scripts

`step5` (copy of **`step5skel`**):

```
ROOTDIR=$PWD/newroot

[ -d "${ROOTDIR}" ] || mkdir "${ROOTDIR}"
mount -n -t tmpfs -o size=50M none "${ROOTDIR}"
rsync -a skelfs/ "${ROOTDIR}"

cd "${ROOTDIR}"

pivot_root . oldroot

mount -t proc proc /proc

export PS1="[\u@\h \W]# "

bash
```

Run **`step1`**:

```
$ sudo ./step1
```

Containers – A look under the hood

Capabilities

Capabilities – introduction

Traditional UNIX privilege scheme

- process running with UID 0 (superuser): *all* privileged actions allowed
  process running with UID ≠ 0: *no* privileged actions allowed

Linux privilege scheme

- *capabilities*
  - collection of distinct privileges that can be set for process (thread) or not
  - examples: **`CAP_CHOWN`**, **`CAP_KILL`**, **`CAP_SYS_BOOT`**,
    `CAP_SYS_TIME`, **`CAP_SYS_NICE`**, …

    see man page **`capabilities(7)`**
  - kernel code always checks on single capabilities, not on UID 0
  - thread running with effective UID 0: initially all capabilities set

Capabilities – process/thread

Each process (thread) has five capability sets

- *effective:* verified for each privileged action
- *permitted:* allowed to be set in effective or inheritable set
- *inheritable*, *ambient* and *bounding* sets outside scope of this presentation

Examples:

own shell

```
$ cat /proc/$$/status
....
CapEff: 0000000000000000
CapPrm: 0000000000000000
CapInh: 0000000000000000
CapAmb: 0000000000000000
CapBnd: 000001ffffffffff
```

systemd

```
$ cat /proc/1/status
....
CapEff: 000001ffffffffff
CapPrm: 000001ffffffffff
CapInh: 0000000000000000
CapAmb: 0000000000000000
CapBnd: 000001ffffffffff
```

Capabilities – scenarios

Possible scenarios

- process with EUID $\neq 0$ (non-superuser)
  - without any capability set (traditional scheme), or
  - with certain or even all capabilities set

- process with EUID $= 0$ (superuser)
  - with all capabilities set (traditional scheme), or
  - with limited capabilities set, or
  - even without any capability set

Capabilities and Docker

Control capabilities with command **`docker run`**

- non-root user in container: no capabilities

- root user in container (also when container started by user in group **`docker`**)
  - default set of capabilities

    `setuid chown net_raw dac_override fowner kill sys_chroot`
    `setgid mknod setpcap audit_write fsetid setfcap net_bind_service`

  - add or drop capabilities (**`all`** for *all* capabilities)

    `docker run --cap-add` ***caps*** **`--cap-drop`** ***caps*** **`....`**

```
$ docker run -it  --cap-drop all --cap-add sys_nice  ubuntu
root@2fd64a4dc104:/# nice -n -11 sleep 100&
root@2fd64a4dc104:/# ps -l
F S   UID   PID  PPID  C PRI  NI ADDR    SZ WCHAN  TTY        TIME CMD
4 S     0     1     0  0  80   0 -     8721 wait   ?      00:00:00 bash
4 S     0    13     1  0  69 -11 -     1130 hrtime ?      00:00:00 sleep
```

  - all privileges without limitations and with unrestricted device access

    `docker run --privileged ....`

Containers – A look under the hood

Questions?