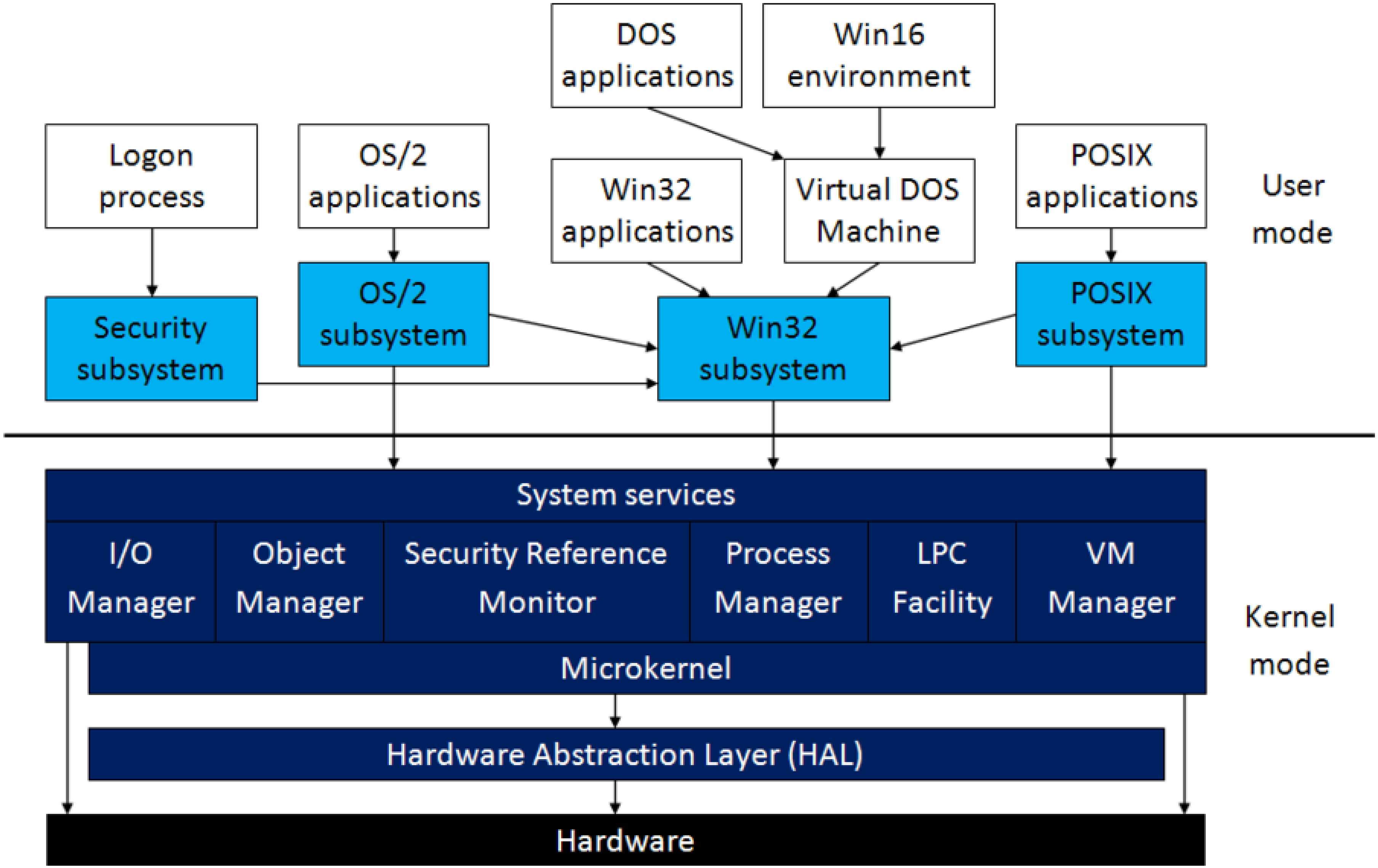
Containers

- Motivation
  - Give each process its own environment
    - Environment variables alone are not sufficient to solve the Dependency hell
      - Incompatible versions of installed libraries
      - Incompatible behavior of installed executables
      - Unexpected system configuration stored in user-accessible files
    - Some applications come from a different ecosystem
      - Different conventions regarding the filesystem
      - Different flavor of the OS
  - Improve isolation between processes
    - Processes may refuse to work with limited privileges
      - Create an illusion that they have privileges they actually have not
    - Avoid conflicts on well-known ports, implant a firewall between local processes
      - Create virtual networks and link processes to virtual NICs
- Linux Containers are not the first attempt
  - At least for some of the goals

Subsystems in Microsoft Windows

Microsoft Windows NT 3.1 (1993)



Containers in Windows

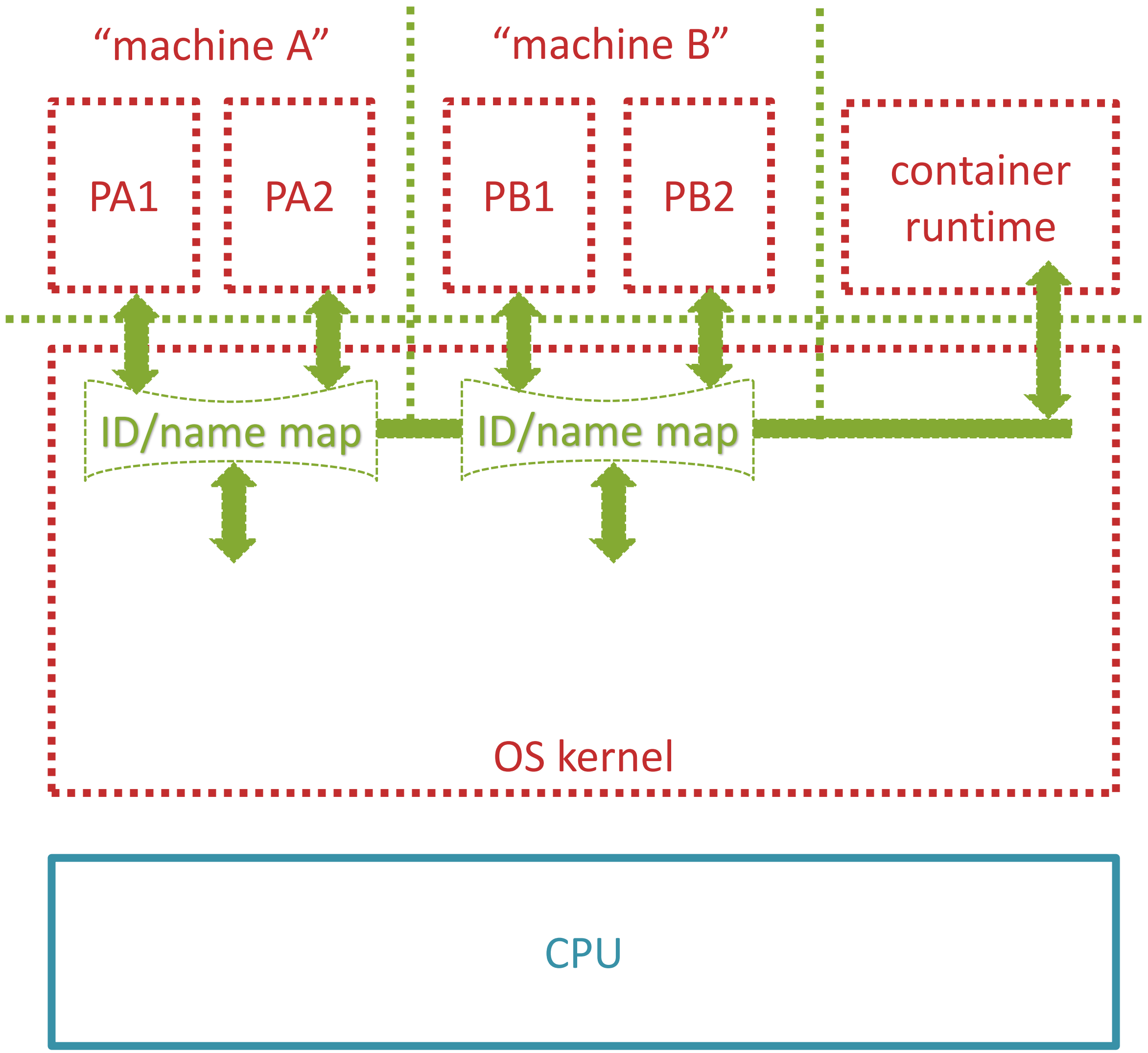
- (Windows) NT kernel was created to support several kinds of apps
  - (IBM) OS/2
  - (Microsoft) Windows 3.1 (binary compatible with non-NT “kernels”)
  - Legacy 16-bit Windows and DOS
  - POSIX
- The NT kernel always included support for namespace isolation and resource limiting
  - In limited use before 2016
- Windows Subsystem for Linux (WSL, bash.exe) – 2016
  - Emulates Linux syscalls on a Windows kernel
    - Does not emulate Linux namespaces and cgroups – cannot support Linux containers
- Windows Containers – 2016
  - Part of the Docker team acquired by Microsoft in 2014
  - Docker-like images and containers for running Windows processes
  - Two modes of container execution
    - Process Isolation – the Windows kernel provides isolation
    - Hyper-V Isolation – each VM runs its own Windows Server kernel

Containers in Windows

- Windows Subsystem for Linux
  - WSL 1 (2016) - Emulates Linux syscalls on a Windows kernel
    - Does not emulate Linux namespaces and cgroups – cannot support Linux containers
    - Uses NTFS – lower performance than Linux, faster sharing with Windows
  - WSL 2 (April 2020) – Runs a true Linux kernel in a Hyper-V virtual machine
    - Can support Linux containers
    - Native unix FS – faster local files, slower access to host Windows files than in WSL 1
- Windows Containers
  - Inside a container, only Windows Server environment is supported
  - Process Isolation - the Windows kernel provides isolation
    - Supported by Windows Server (since 2016), Windows 10 (since April 2020)
  - Hyper-V Isolation – each VM runs its own Windows Server kernel
    - Supported by Windows Server (since 2016), Windows 10 (since September 2018)
  - May be managed by Azure versions of Docker, Kubernetes, etc.
    - Management almost identical to Linux containers (when run inside Azure)
  - Not nearly as successful as Linux containers
    - 28K Windows vs. 3.5M Linux containers on hub.docker.com (October 2020)

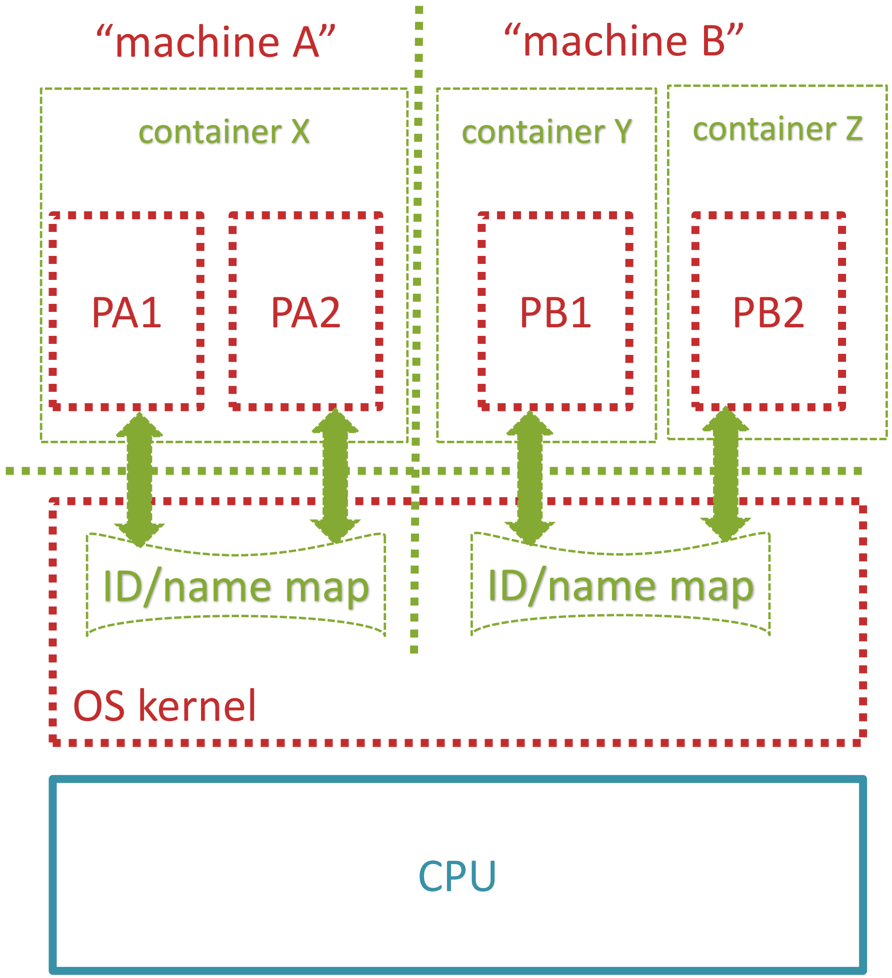
Containers
(Linux)

Containerization



- Namespace separation
  - The upper layer of the OS kernel filters the syscalls and maps all the identifiers from process-specific to system-wide naming spaces
- Resource separation
  - The kernel maintains resource usage statistics for each set of processes and restricts them
- Container runtime
  - Optional
  - Privileged process used to setup the kernel maps and react to events

Containerization – machines vs. containers



- Container (simplified definition)
  - a file system plus a configuration
  - when started, a configured command is executed
    - it starts an executable from the internal file system
    - this executable may later spawn more processes (via fork/exec/system)
  - a running container may contain more than one process
- OS kernel can map several containers to the same system resources
  - podman **pod** = set of containers
    - all containers in a pod share the same NIC (and some other namespaces)
    - each container has its own filesystem
- Some container systems allow direct access to host NIC
  - no virtual network/NAT = faster
  - decreased safety and isolation

Linux namespaces

- Linux namespaces
  - A namespace defines the mapping of identifiers
    - from the local view of the process
    - to the global identifiers used inside the kernel
    - applied on each SYSCALL to translate local ids to global and back
    - it may also define how new ids are created
    - some namespaces (NET, CGROUP) also configure the behavior of the kernel
- cgroups
  - A cgroup defines a unit of accounting
    - Processes in a cgroup share the same pool of resources
    - A cgroup may also define a policy applied by the kernel
- USER and PID namespaces and all cgroups form hierarchies
    - The root namespace is the 1:1 mapping applied to the *init* process and others
    - The root cgroup represents all the resources of the machine and kernel
    - Child namespaces/cgroups are subsets of their parents, with elements renamed
- Other kinds of namespaces are not hierarchical
    - Their elements may be unreachable from other namespaces

Linux namespaces

- The most important types of namespaces (in the order of appearance)
  - Mount - mounts, i.e. the complete filesystem
    - Linux 2.4.19 – August 2002
  - UTS - machine name, OS version, etc.
    - Linux 2.6.19 – November 2006
  - IPC - ids of message queues, semaphores, shared memory
    - Linux 2.6.19 – November 2006
  - USER - user and group ids (numeric)
    - Linux 2.6.23 – October 2007
    - changed semantics in Linux 3.5 - Jul 2012, finished in Linux 3.8 - Feb 2013
  - PID - process and thread ids (numeric)
    - Linux 2.6.24 – January 2008
  - Network - the complete configuration of networking (NICs, ports, routing, forwarding)
    - Linux 2.6.29 – April 2009
  - Cgroup - resource-sharing pool and the associated cgroup configuration
    - Linux 4.6 – May 2016
  - Time - adjustments to monotonic clock (to make container migration possible)
    - Linux 5.6 - March 2020

Linux namespaces

- cgroup version 1 was abandoned, version 2 is now in use
- a cgroup is a set of *controllers* and their configuration
  - io – accessible bandwidth of block device I/O (since Linux 4.5)
  - memory – process/kernel/swap memory (since Linux 4.5)
  - pids – max number of processes/threads created (since Linux 4.5)
  - perf_event – performance monitoring (since Linux 4.11)
  - rdma – access to DMA resources in the kernel and the hardware (since Linux 4.11)
  - cpu – CPU time allotment (since Linux 4.15)
  - cpuset – set of CPU or NUMA nodes available (since Linux 5.0)
  - freezer – suspending/restoring all processes in a cgroup (since Linux 5.2)
  - hugetlb – allocation of huge TLB pages (since Linux 5.6)
- other features attached to a cgroup
  - access to I/O devices
  - packet filtering may be based on the id of the originating cgroup

Process (Linux)

- A Linux process consists [mainly] of
  - **pid**, parent **pid**
  - effective **uid**, **gid**, *capabilities*, etc.
  - attached *namespaces* (one namespace per each type of namespace)
  - file descriptors (open files, pipes, semaphores, etc.)
  - virtual memory
  - state, CPU registers
- Processes are created by syscalls:
  - **fork** – copy everything (except pid/parent pid and the return value from fork)
  - **clone** – each of the constituents may be shared or copied or created new
    - behavior controlled by flags
    - example: sharing everything (except CPU registers) creates a thread
- The **exec** syscall is the only way to load an executable file
    - it replaces actual virtual memory with the new code and data, resets state
    - effective uid/gid/capabilities may change if the executable file has **suid** bit set

- Linux namespaces are created by these syscalls:
  - **clone** – for the namespace types selected by flags, new namespaces are created for the child process (the other types are shared)
  - **unshare** – for the namespace types selected by flags, new namespaces are attached to the calling process (the previous namespaces are detached but continue to exist)
  - The new namespaces
    - set as ***owned*** by the user namespace that
      - was created by the same syscall (if there was one)
      - was attached to the calling process before the syscall (otherwise)
    - user and pid namespaces are permanently set as ***children*** of the namespaces of the same type attached to the calling process before the call
    - the contents of the new namespaces after clone/unshare:
      - user, network, and ipc namespaces are empty
      - after clone, pid namespaces contain the newly created process with pid=1
      - other namespace types (mount etc.) are copies of their parents

- Namespace is discarded when
  - No attached processes exist
  - No child namespaces exist (for user and pid namespaces)
  - No owned namespaces exist (for a user namespace)
  - No bind mount exist that represents the namespace
    - Namespaces are represented by /proc/<pid>/ns/* virtual files, these may be duplicated by bind-mounting elsewhere

- Setting the contents of the new namespaces
  - may be performed by processes attached to
    - the parent namespace of the same type
    - the same namespace
  - usually performed between **clone/unshare** and **exec** calls, i.e. by the same code that called clone/unshare
    - this code is aware of both the existing parent and the desired child identifiers
  - often performed by manipulating /proc/<pid>/* files
    - other, namespace-specific ways exist (e.g. the MOUNT syscall)

Linux procfs

- **procfs** filesystem (since 1984)
  - usually mounted at /proc
    - the contents reflects the pid namespace of the process that called mount
    - must be mounted again inside a container
  - contains virtual folders and files
    - enables communication between the kernel and user processes
      - reduces the number of syscalls required
      - allows passing more than the 6 64-bit parameters/results of a syscall
    - any access to /proc/* is done using universal OPEN/READDIR/READ/WRITE syscalls
      - standard mechanism of file access rights applies
      - READ/WRITE have a mechanism for large data transfers between process and kernel
  - in procfs, each filename has its own READ/WRITE handler
    - READ converts some kernel data to file contents, often in tab-separated decimal form
    - WRITE (if enabled) analyzes the text and sets the kernel data
      - often limited to single OPEN-WRITE-CLOSE syscall sequence
    - disadvantage: the kernel contains code for producing/parsing text and numbers
  - majority of the contents (but not all) presented as /proc/<pid>/*
    - some folders/files are presented relative to the calling process, e.g. /proc/self
  - example: the **ps** utility works by reading the virtual files in /proc

Linux namespaces – capabilities

- Each process has a bit mask of (about 40) capabilities
  - A fine-grained replacement (since 1999) for testing effective uid==0
    - However, majority of privileged actions are still controlled by the CAP_SYS_ADMIN capability
  - The capabilities are bound to the user namespace attached to the process
    - Applicable to actions on and in namespaces *owned* by this user namespace
  - The process that enters (by clone/unshare) a newly created user namespace
    - Automatically holds all capabilities (wrt. this user namespace)
    - It may propagate these capabilities to child processes
    - It will loose the capabilities on exec, unless its effective uid (in its namespace) is zero
- User namespaces
  - Any process can create a user namespace
  - CAP_SETUID in the *parent* user namespace is required to setup a non-trivial user mapping
    - CAP_SETUID normally allows impersonation of anyone in the same namespace (e.g. by sshd)
    - the impersonation can also happen by mapping a user from a child user namespace
  - non-CAP_SETUID-equipped processes can only setup a trivial user mapping
    - map one (arbitrary) child uid to the effective uid of the process that created the namespace
- Non-user namespaces
  - CAP_SYS_ADMIN is required to create a non-user namespace
    - if a new user namespace is created by the same call, the capability is automatically assumed
    - otherwise, the invoking process must have had that capability before
  - A specific capability is required when
    - The id mapping associated with a namespace is defined (e.g. pid generators)
    - Objects in the namespace are created (e.g. network devices) or modified (e.g. firewall rules) in such a way that may affects all processes in the namespaces

Linux namespaces – mapping uids and gids

- Technically, uid and gid mapping is limited to a (small) set of intervals of uids/guids mapped linearly from the child to the parent
    - The mapping is defined by writing /proc/<pid>/{uid_map|gid_map}
  - Unmapped child-namespace uids/gids cannot be used in any syscall (like setuid or chown)
  - Unmapped parent-namespace uids/gids (e.g. from a file system) cannot be presented to processes in the child namespace
    - Mapped as 65534 (usually decoded as "nobody" by /etc/passwd and /etc/group)
- Non-privileged processes may directly map only one child uid/gid
  - This child uid/gid may be 0 ("root")
  - It must be mapped to the effective uid/gid of the process that created the user namespace
- Indirect setup using **newuidmap** and **newgidmap** utilities
  - Available to any user for any user namespace created by this user
  - These executables have CAP_SETUID capability attached and may therefore setup arbitrary uid/gid mappings
  - However, these utilities allow only mappings that
    - Map at most one child uid/gid to the uid/gid of the calling user
    - All the other child uid/gid must map into the range(s) defined for the calling user by the /etc/subuid and /etc/subgid files
    - In default settings, each standard user has 65536 additional uids and gids reserved by the /etc/sub*id files
    - The rules ensure that different standard users can never use the same parent uids/gids
    - The additional uids/gids are not present in the (parent mount namespace) /etc/passwd or /etc/groups; therefore, they are displayed numerically by utilities like ls

- **unshare** utility can launch a new process into new namespaces
  - Namespace creation controlled by command-line options
- User namespace - trivial mapping to self

```
[bednarek@rocky ~]$ unshare -c
```

  - The above command launches bash into a new user namespace

```
[bednarek@rocky ~]$ ps -l
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  344957  344929  0  80   0 -  2267 -      pts/3    00:00:00 bash
4 S  1000  350824  344957  0  80   0 -  2265 do_wai pts/3    00:00:00 bash
0 R  1000  350881  350824  0  80   0 -  2521 -      pts/3    00:00:00 ps
```

    - This namespace has trivial mapping of the current UID/GID to itself

```
[bednarek@rocky ~]$ cat /proc/$$/uid_map
      1000       1000          1
```

    - There is no new mount namespace - we can see the global filesystem

```
[bednarek@rocky ~]$ ls -ld /home/bednarek
drwx------. 15 bednarek bednarek 4096 Oct 25 10:27 /home/bednarek
```

    - However, unmapped global UIDs/GIDs are shown as nobody

```
[bednarek@rocky ~]$ ls -ld /root
dr-xr-x---. 5 nobody nobody 4096 Sep 20 22:56 /root
```

Linux namespaces – unshare utility

- User namespace - trivial mapping of local root to global self

```
[bednarek@rocky ~]$ unshare -r
```

  - All the global user's processes are now shown with local UID=0
    - We can see the parent bash because there is no new PID namespace

```
[root@rocky ~]# ps -l
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S     0  344957  344929  0  80   0 -  2267 -      pts/3    00:00:00 bash
4 S     0  351664  344957  0  80   0 -  2265 do_wai pts/3    00:00:00 bash
0 R     0  351707  351664  0  80   0 -  2521 -      pts/3    00:00:00 ps
```

    - This namespace has trivial mapping of local 0 to the global UID/GID of the user

```
[root@rocky ~]# cat /proc/$$/uid_map
         0       1000          1
```

    - This user's files are now shown as owned by (local) root
      - Actually, this is local UID/GID 0 incorrectly mapped through the global /etc/{passwd,group}

```
[root@rocky ~]# ls -ld /home/bednarek
drwx------. 15 root root 4096 Oct 25 10:27 /home/bednarek
```

    - The true global root is shown as nobody

```
[root@rocky ~]# ls -ld /root
dr-xr-x---. 5 nobody nobody 4096 Sep 20 22:56 /root
```

```
[bednarek@rocky ~]$ unshare -U
```

- Creates a new user namespace with no mapping

```
[nobody@rocky ~]$ cat /proc/$$/uid_map
```

  - Even the actual user is mapped to UID=65534 (nobody)

```
[nobody@rocky ~]$ ps -l
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S 65534  344957  344929  0  80   0 -  2267 -      pts/3    00:00:00 bash
0 S 65534  352808  344957  0  80   0 -  2265 do_wai pts/3    00:00:00 bash
0 R 65534  352872  352808  0  80   0 -  2521 -      pts/3    00:00:00 ps
```

- The mapping must be defined from the parent process
    - We need the SETUID capability in the global user namespace
  - We can map only to global UIDs/GIDs defined by /etc/{subuid,subgid}

```
[bednarek@rocky ~]$ grep bednarek /etc/subuid
bednarek:100000:65536
```

  - The SETUID capability is attached to the newuidmap/newgidmap utilities

```
[bednarek@rocky ~]$ newuidmap 352808 0 1000 1 1 100001 999
[bednarek@rocky ~]$ newgidmap 352808 0 1000 1 1 100001 999
```

- Back in the local namespace, the new maps are visible

```
[nobody@rocky ~]$ cat /proc/$$/uid_map
         0       1000          1
         1     100001        999
[nobody@rocky ~]$ ls -ld /home/bednarek
drwx------. 15 root root 4096 Oct 25 10:27 /home/bednarek
```

```
[bednarek@rocky ~]$ unshare -U
```

- Creates a new user namespace with no mapping
- The mapping must be defined from the parent process
- Back in the local namespace, the new maps are visible

```
[nobody@rocky ~]$ cat /proc/$$/uid_map
         0       1000          1
         1     100001        999
```

  - Note: The "nobody" is still here because the bash was not told to update the prompt

- We can now use all local UIDs between 0 and 999

```
[nobody@rocky ~]$ mkdir test
[nobody@rocky ~]$ chown mail:mail test
```

  - We can execute **chown** because we are local UID=0 and have the local SETUID capability

```
[nobody@rocky ~]$ ls -ld test
drwxr-xr-x. 2 mail mail 6 Oct 25 11:18 test
```

  - Again, "mail" is mapped through global /etc/{passwd,group} to local UID=8, GID=12

```
[nobody@rocky ~]$ grep mail /etc/{passwd,group}
/etc/passwd:mail:x:8:12:mail:/var/spool/mail:/sbin/nologin
/etc/group:mail:x:12:postfix
```

- In the global namespace, the folder is seen with the global UID/GID

```
[bednarek@rocky ~]$ ls -ld test
drwxr-xr-x. 2 100008 100012 6 Oct 25 11:18 test
```

  - If the local UID=8, GID=12 were not mapped, the **chown** above would have failed

Linux namespaces – root-full vs. root-less containers

- Root-full container
  - The initial process of the container runs with uid/gid == 0 (as seen inside the container)
    - It also has all capabilities (wrt. objects in its namespaces)
  - Created by root (sudo) user (of the parent namespace)
    - 1:1 uid/gid mapping or no user namespace at all
    - Dangerous, the only scenario available in the past
  - Created by a non-privileged user
    - uid/gid 0 in the container maps to the creator user/group
    - other uids/gids in the container (if any) map to the creator's subuid/subgid set
- Root-less container
  - All the processes of the container run with the same uid/gid != 0
    - They have no capabilities (therefore unable to create/impersonate other uids/gids)
  - Created by root (sudo) user (of the parent namespace)
    - The only uid/gid mapped to a selected user/group
  - Created by a non-privileged user
    - The only uid/gid mapped to the creator user/group

Containers (Linux)

- The namespaces and cgroups are relatively old mechanism of the kernel
- Some parts were significantly redefined only recently
  - PIDS, capabilities, ...
- Many container systems use older, less general kernel mechanisms
  - Instead of using the mechanism of owner namespaces, docker does this:
    - *docker* executable forwards the commands via a named pipe to the *dockerd* daemon
    - *dockerd* daemon uses root privileges to manipulate the namespaces and cgroups
  - Consequently, the safety of the system relies on the correctness of *dockerd*
- *Red Hat* reacted by implementing *podman*, which implements docker commands through the modern kernel mechanisms, bypassing any daemon

Containers (Linux)

- There are conflicting philosophies with respect to containers
  - Docker, Inc.: Containers are lightweight entities
    - A container shall typically contain only one process
    - Any connection between processes shall be handled outside the containers
      - Use Kubernetes to orchestrate these connections
    - To update the software in a container, drop the container and start another
      - Due to robustness and load-balancing requirements, the container must survive this anyway
  - Red Hat, Inc.: Containers are like computers
    - Many applications consists of several processes
      - apache, mysql, java, cron, ...
    - The applications are published with a sophisticated installation script
      - Nobody is going to rewrite installation scripts into Kubernetes configurations
    - Installation scripts shall work inside containers
    - Typical installation procedures shall work inside containers:

```
$ sudo yum install gcc
$ sudo yum upgrade
$ sudo systemctl enable sshd
```

Containers (Linux)

- PID namespace
  - This happens in a lightweight container *without* pid namespace, executing "bash":

```
# systemctl status
Failed to connect to bus: Operation not permitted
# sudo systemctl status
sudo: /etc/sudo.conf is owned by uid 65534, should be 0
sudo: /etc/sudo.conf is owned by uid 65534, should be 0
sudo: error in /etc/sudo.conf, line 0 while loading plugin "sudoers_policy"
sudo: /usr/libexec/sudo/sudoers.so must be owned by uid 0
sudo: fatal error, unable to load plugins
# ls /etc/sudo.conf -ln
-rw-r-----. 1 65534 65534 1786 Apr 24  2020 /etc/sudo.conf
# grep root\\\|65534 /etc/passwd
root:x:0:0:root:/root:/bin/bash
nobody:x:65534:65534:Kernel Overflow User:/:/sbin/nologin
```

  - The process PID=1 has two special roles
    - it controls daemons – published via a named pipe as the *systemctl* command
    - it collects zombies
  - Inside a typical container, PID=1 is the main executable, often a shell
    - it cannot respond to the systemctl request
  - *sudo* refuses to work because the true owner of sudo.conf does not exist inside the USER namespace of the container
  - the *root* of the container namespace is not configured to have sufficient privileges

Linux namespaces – unshare utility - pid namespace

- Creating a new pid namespace - unsuccessful attempts

```
[bednarek@rocky ~]$ unshare -p
unshare: unshare failed: Operation not permitted
```

  - Creating any namespace other than user namespace requires CAP_SYS_ADMIN
  - We can acquire this capability by entering a new user namespace (here with -r)

```
[bednarek@rocky ~]$ unshare -r -p
-bash: fork: Cannot allocate memory
-bash-5.1# echo $$
373218
```

  - A pid namespace requires a really new process, not just unsharing

```
[bednarek@rocky ~]$ unshare -r -p --fork
basename: missing operand
Try 'basename --help' for more information.
[root@rocky ~]# echo $$
1
```

  - We are in the new pid namespace with PID=1

```
[root@rocky ~]# ps
    PID TTY          TIME CMD
 344957 pts/3    00:00:00 bash
 373102 pts/3    00:00:00 unshare
 373103 pts/3    00:00:00 bash
 373148 pts/3    00:00:00 ps
```

  - But ps is implemented using /proc, so we actually see the global processes
  - Our bash with local PID=1 maps to global PID=373103

- Creating a new pid namespace - the correct way

```
[bednarek@rocky ~]$ unshare -r -p --fork --mount-proc
```

  - The --mount-proc switch mounts a new instance of procfs to /proc
  - Before that, the utility created a new mount namespace

```
[root@rocky ~]# echo $$
1
```

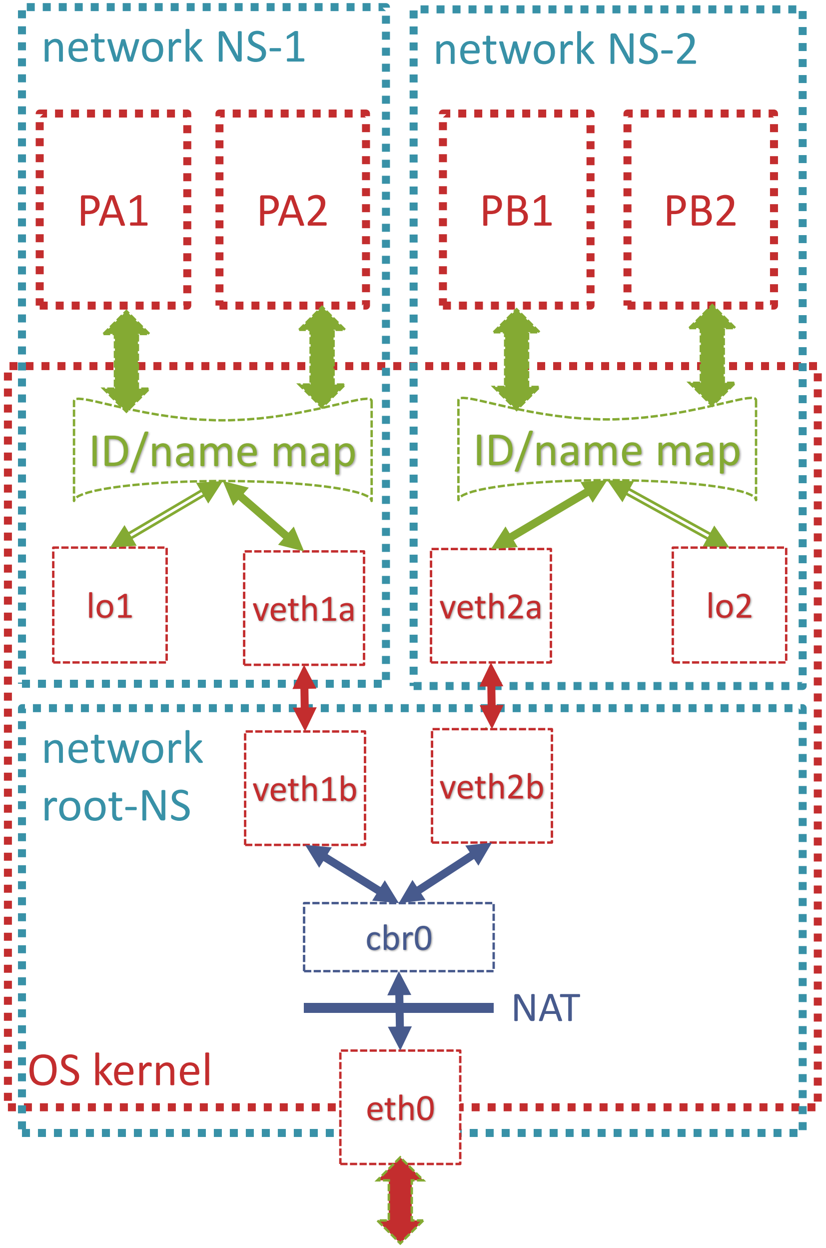
  - Our bash is running with local PID=1

```
[root@rocky ~]# ps -el
F S   UID     PID    PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
4 S     0       1       0  0  80   0 -  2265 do_wai pts/3    00:00:00 bash
0 R     0      33       1  0  80   0 -  2521 -      pts/3    00:00:00 ps
```

  - We can't see any other processes than the PID=1 and the ps utility itself

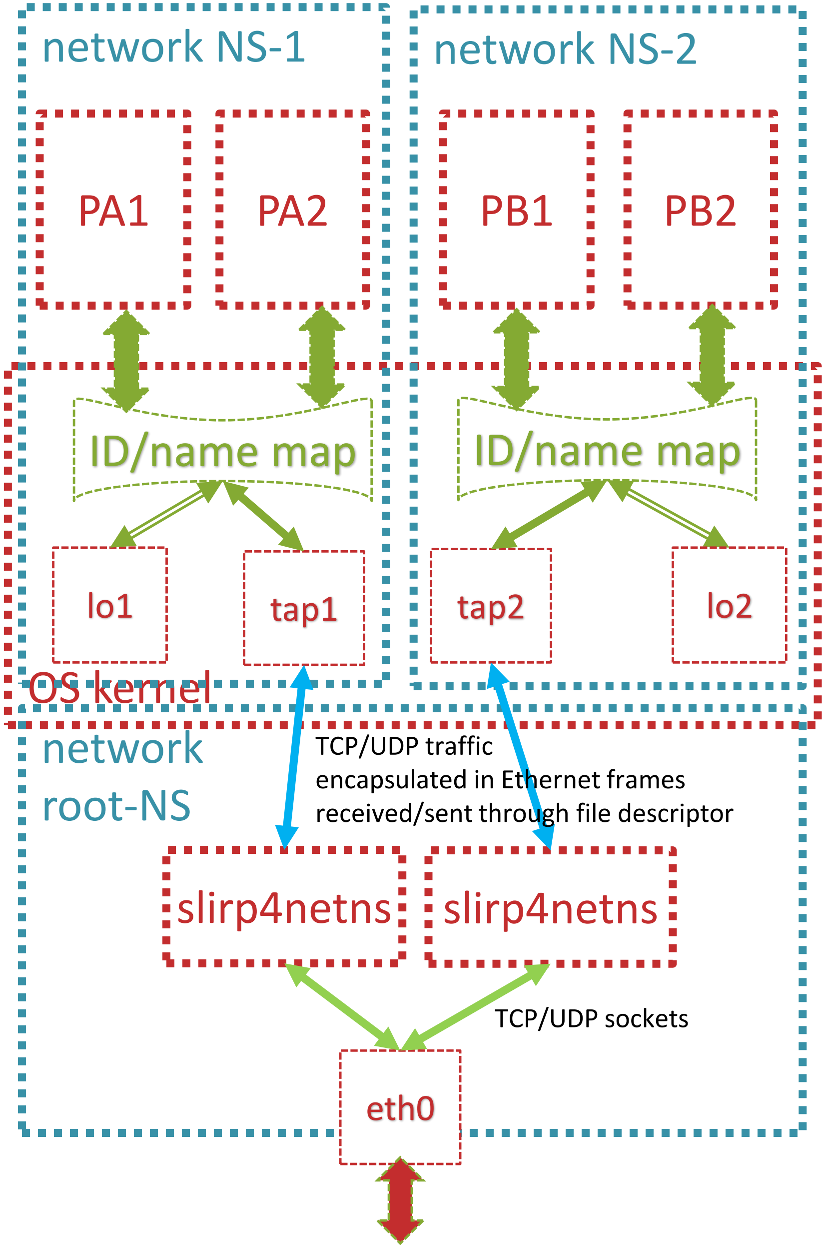
- This is the minimum that a modern container system must do
  - At least when system container (with PID=1 and UID=0) is required
- Create a user namespace and map UID=0 to the parent user
- Create a mount namespace
  - Real containers would map their own filesystems here
- Fork a new process into a new pid namespace
  - Mount a new procfs into /proc
- Real containers usually also create a network namespace

Containerization – network namespaces



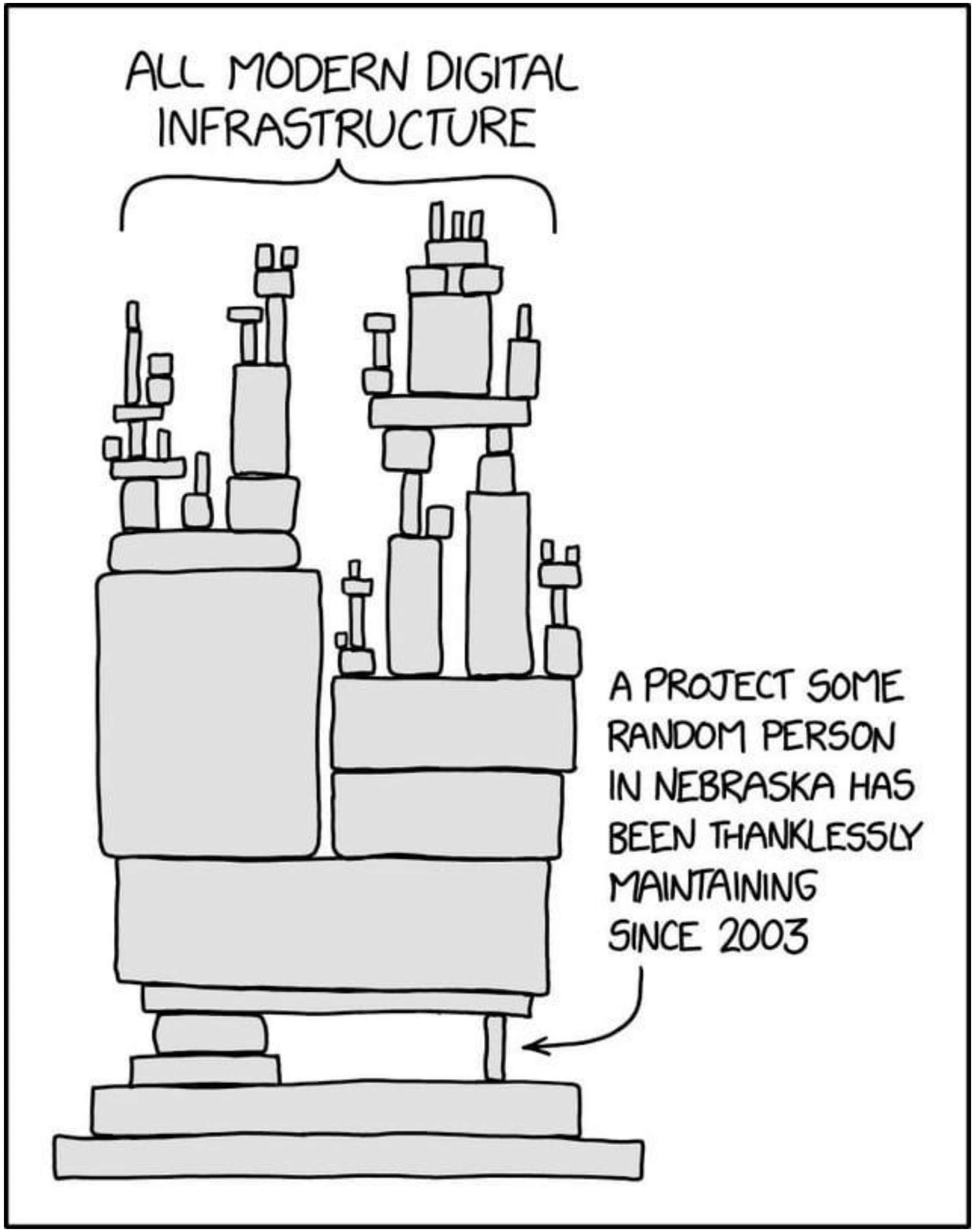
- Network namespaces are created empty
  - Devices, routing and firewall rules are bound to a NS
- veth – a pair of virtual Ethernet devices
  - packets sent through one side are received on the other
  - usually installed across network NS boundary
    - privileges required in both namespaces
      - non-root users must provide network access differently
- The outer side of the veth pair
  - Usually connected to a virtual bridge
    - More than one container may reside in the virtual LAN
      - Example: podman pod
    - Unrestricted connections between such containers
      - Restrictions may be set by firewall rules in NSs
  - Router mode
    - Host linux kernel (root network NS) acts as the router
    - Routing with NAT (usually the default)
      - Containers have private addresses
      - External access requires port forwarding
    - Routing without NAT
      - Containers have public addresses
      - External access may be blocked by host firewall
  - Bridge mode
    - Host physical network is attached to the bridge
      - Containers have public addresses
      - No routing/firewall provided by the host
      - Non-IP LAN connectivity may be provided

Containerization – network namespaces for non-privileged creators



- Network namespaces are created empty
  - Devices, routing and firewall rules are bound to a NS
  - Non-privileged creator cannot create a veth pair
    - due to insufficient privilege in the root NS
  - Non-privileged creator can create a TAP adapter
    - using root privileges in the child NS
    - the TAP adapter is connected to user-space stack
  - slirp4netns
    - an utility developed from slirp (1996)
      - not seriously secure!
    - receive/send Ethernet packets via a TAP
    - send/receive unencapsulated TCP/UDP traffic
      - using unprivileged TCP/UDP ports
      - cannot use port < 1024
    - in effect, similar to a NAT router
      - but implemented quite differently
      - no container-to-container traffic
    - root-less container systems invoke this daemon automatically

xkcd 2347


ALL MODERN DIGITAL INFRASTRUCTURE
A PROJECT SOME RANDOM PERSON IN NEBRASKA HAS BEEN THANKLESSLY MAINTAINING SINCE 2003

- The userspace layer of containers
  - docker, podman, ...
  - An *image* is essentially a **read-only** filesystem
    - Plus some defaults and interface declarations
  - A *container* is an image plus
    - A **writable** layer above the image filesystem
      - This is destroyed when the container is deleted (but survives stops)
    - A set of mounts used to access some folders outside the container
      - This can survive deleting and recreating the container (e.g., from an updated image)
    - A set of ports mapped via virtual networks to the outside world
  - A *running container* is
    - A set of processes living in the namespace of the container
      - Created by forking from a single process, usually the *ENTRYPOINT* defined in the image
    - Optionally, stdin/stdout/stderr pipes attached to the processes

Containers (Linux)

- The image is created by adding layers
    - To another image or to an empty filesystem ("FROM SCRATCH")
- Each layer can be
    - A set of files copied from elsewhere
        - e.g., docker can download and unzip something
        - This way the underlying Linux distro is applied
    - The result of a command executed inside the container
        - A writable filesystem layer is added to the container
        - The command is executed inside an environment similar to a container
            - Usually inside a restrictive namespace but without port remapping
        - When done, the writable layer is frozen to read-only
- The layers are combined using a kind of *union filesystem*
    - A filesystem combining two other filesystems (e.g. overlayfs)
        - *Whiteout*: deleting in the upper filesystem hides a file from the lower filesystem
    - The container manager may reuse a layer in more than one image/container
        - If the layers were created by the same commands or have the same hash
        - Significantly lowered disk space consumption (but hardly predictable)

Containers (Linux)

- The layers are combined using a kind of *union filesystem*
  - A filesystem combining two other filesystems (e.g. overlayfs)
- Each layer may be
  - A subtree of a physical (host) file system
  - A separate file system over a virtual block device
    - Usually implemented in a binary file
- Overlay FS, layer filesystems and virtual block devices
  - Implemented in kernel when set up by privileged users
    - Permissions and owner UID/GIDs stored within FS
      - Container images cannot be shared between different host users
  - Implemented in userspace when set up by root-less users
    - Using Linux FUSE - FS requests redirected from kernel to user processes
    - Permission checking delegated to the userspace component
      - Container images may be shared if the layer FS is container-aware
- Layers may be flattened into one before running the container
  - Used in performance-oriented container systems (e.g. charliecloud)

Dockerfile

```
# syntax=docker/dockerfile:1
FROM python:latest
WORKDIR /data
ENV TZ=Europe/Prague
ENV FLASK_APP=app.py
ENV FLASK_RUN_HOST=0.0.0.0
COPY code/requirements.txt requirements.txt
RUN pip install -r requirements.txt
VOLUME ["/data"]
EXPOSE 5000
EXPOSE 9876/udp
CMD ["bash", "run_both.bash"]
```

- Dockerfile
  - script to create a container image
    - placed at the source folder
  - direct filesystem modifications
    - FROM - base image
    - COPY - copy from source folder
  - indirect filesystem modifications
    - RUN
      - create a writable layer on top
      - run the specified command in WORKDIR
      - freeze the writable layer
  - setting startup process
    - ENV – process environment
    - CMD/ENTRYPOINT - command
  - metadata
    - VOLUME – mount points
    - EXPOSE – port list

docker

- docker build
  - read Dockerfile and other files
  - pull base image from a **registry**
  - produce container **image**
- docker image push/pull
  - push/pull image to/from a registry
- docker create
  - create a writable layer above an image
  - link mount points as specified
  - connect ports as specified
  - the result is a **stopped container**
- docker start
  - start the startup process
- docker exec
  - implant another process into the container namespaces
- docker stop/kill

- image
  - a combined filesystem
    - sequence of layers (binary blobs)
    - multiple images may share (lower) layers if created by the same commands
  - environment, startup command, mounts, ports
  - created by freezing a container
- container
  - similar to an image
    - the top filesystem layer is writable
  - may be **running** as a subtree of processes
  - namespaces and cgroups ensure the required execution environment

Containers and the outside world

- Mount-points (VOLUME)
  - When started, the internal mount-points are linked to files/folders on the host
    - Specified by options for *docker create* etc.
  - Main purpose: Long-term persistency of data
    - Software in containers is usually updated by creating a new container from an updated image
      - The updated image may be created from the same Dockerfile
      - FROM and RUN commands may produce different outcome
      - The writable layer of a container cannot be reattached to different underlying image
- Ports (EXPOSE)
  - Usually, each container has its own virtual NIC (usually called Bridge mode)
    - When started, the internal ports (associated to a virtual NIC of the container) are linked (via NAT) to the specified host NIC ports
      - Specified by options for *docker create* etc.
  - Alternatively, the container may directly use the host NIC (deprecated)
  - More complex arrangements may exist (not directly exposed by docker)
- IPC
  - Host's named pipes, devices etc. may be exposed to the container
  - stdin/stdout/stderr of the container may be connected to host

docker-compose

```yaml
version: "3.9"
 services:
  web:
   build: .
   ports:
    - "5500:5000"
    - "9876:9876"
   volumes:
    - "./my_data:/data"
   environment:
    FLASK_ENV: development
   image: "repository.local:5555/thermocont"
```

- docker-compose
  - Built above docker
  - Config: docker-compose.yml
  - Building and testing containers
  - Repository operations
  - Combining more containers together (services)