A practical course on

Advanced systems programming in C/Rust

Jörg Thalheim

Today's topic!

Containers

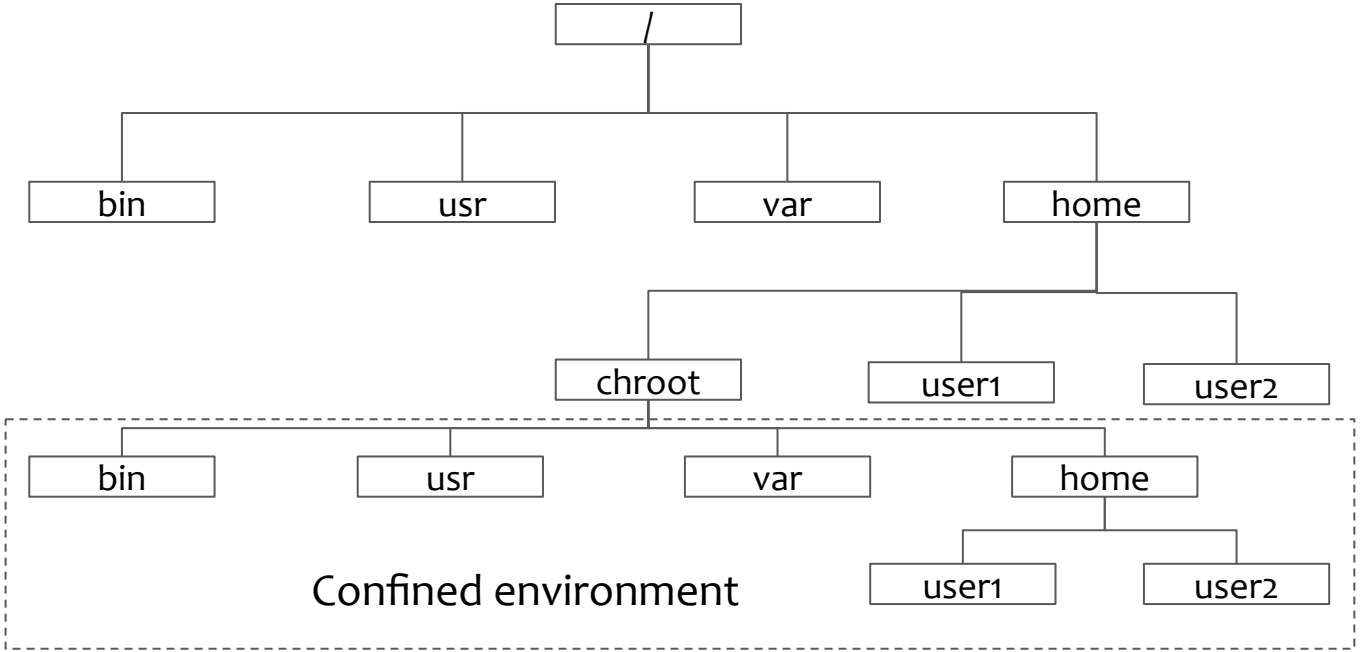
OS-Virtualisation

- Resource isolation only enforced in software - the operating system
    - Granularity: Process level
    - Only one OS unlike hardware virtualisation
    - Flexible resource sharing between processes
- Present in all general purpose operating system:

Unix “chroot”

Chroot: Origins of Containers

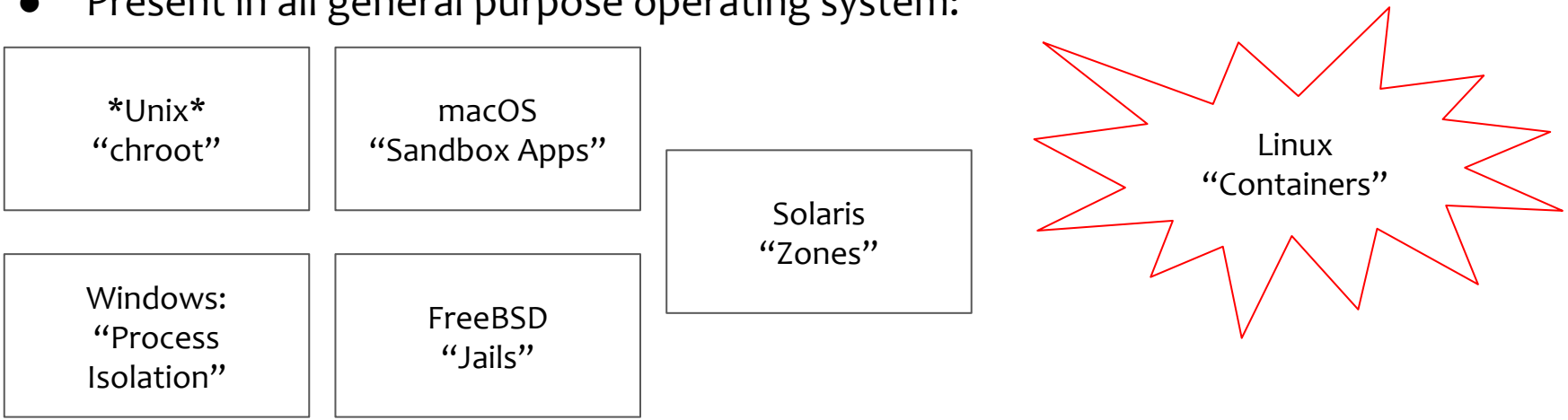
- **Ch**ange **R**oot -> makes a subdirectory the new root of a process
- syscall: chroot()
- Useful to contain unprivileged processes (i.e. postfix mail demon, sshd)
- Root users can easily undo this



OS-Virtualisation

- Resource isolation only enforced in software - the operating system
  - Granularity: Process level
  - Only one OS unlike hardware virtualisation
  - Flexible resource sharing between processes
- Present in all general purpose operating system:

Our focus



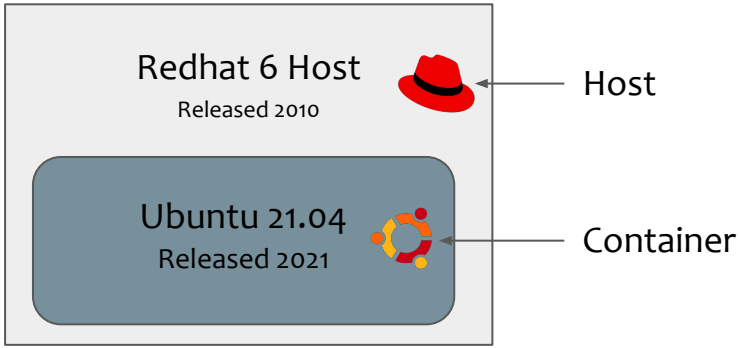
OS-Virtualisation on Linux: Container

Why did it became so popular on Linux?

- (Very) stable syscall interface
  - Kernel devs: “Don't Break Userspace”

  -> libc can be replaced & distributed independently (unlike i.e. FreeBSD jails)

- Convenient application packaging & distribution: Docker -> Kubernetes:
  - Less dependency hell & rollbacks

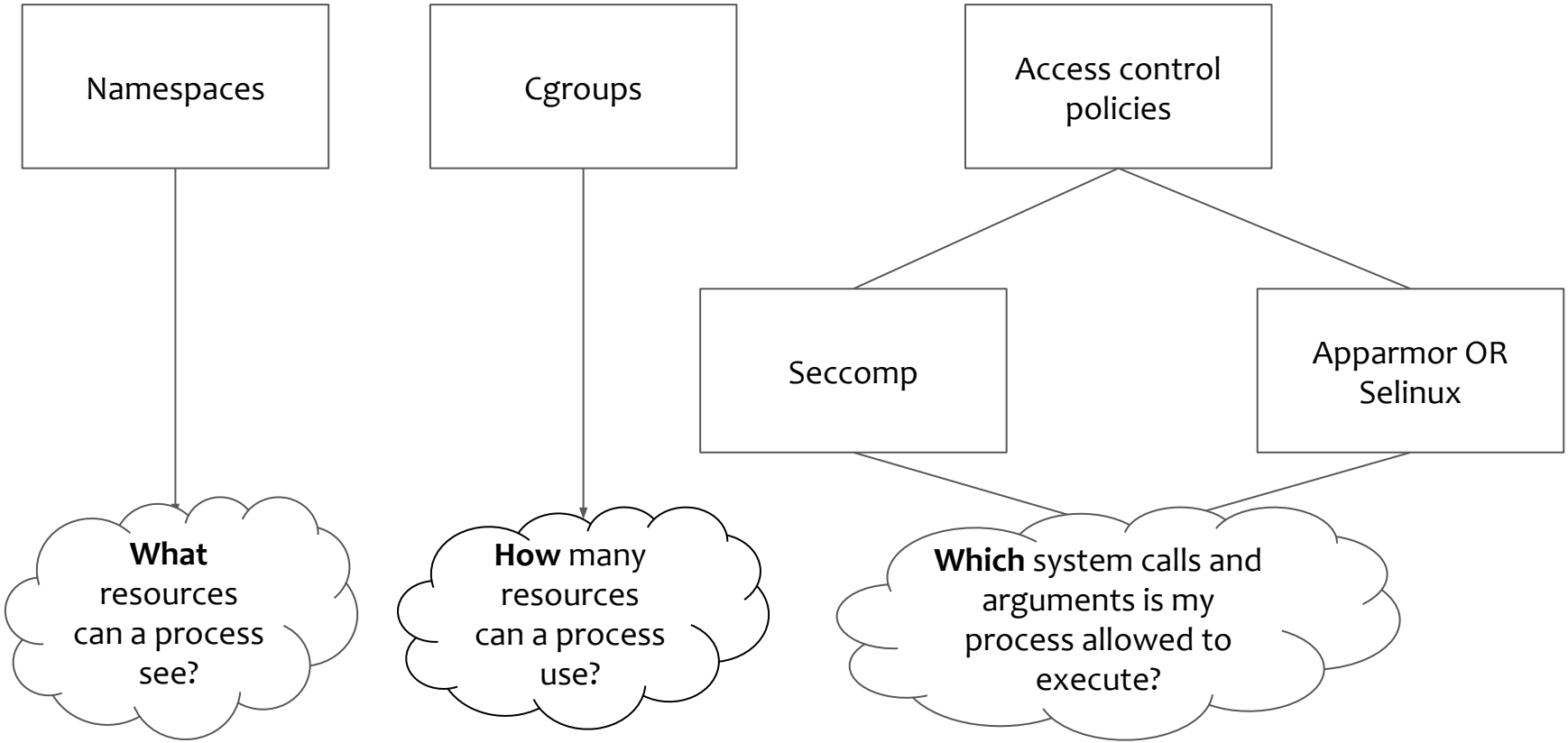


docker

kubernetes

Linux APIs

Linux APIs


Namespaces
Cgroups
Access control policies
Seccomp
Apparmor OR Selinux
What resources can a process see?
How many resources can a process use?
Which system calls and arguments is my process allowed to execute?

Namespaces

- Control visibility of kernel resources (mounts, network interfaces, time etc)
- System calls and kernel subsystems needs to be aware of namespaces:
  - Stored in process context: task_struct->nsproxy
- Pid 1 (init process) starts with initial namespace
- Children inherit namespace from their parents

Create namespace
System call: *unshare()* or *clone()*
Example: $ docker run

Switch namespace
System call: *setns()*
Example: $ docker exec

Inspect namespace
Interface: /proc/self/ns
Example: $ ls -la /proc/self/ns

See “man namespaces” for further information (well written)

Namespaces types

- 8 namespaces so far, but list is growing:
  - Cgroup, IPC, Network, Mount, PID, Time, User, UTS

Mount namespace

- Isolates mountpoints seen in a namespace
- Useful in combination with chroot

PID

- Isolates what processes are visible
- Allows containers to have their own Init process (Pid1)

Network namespaces

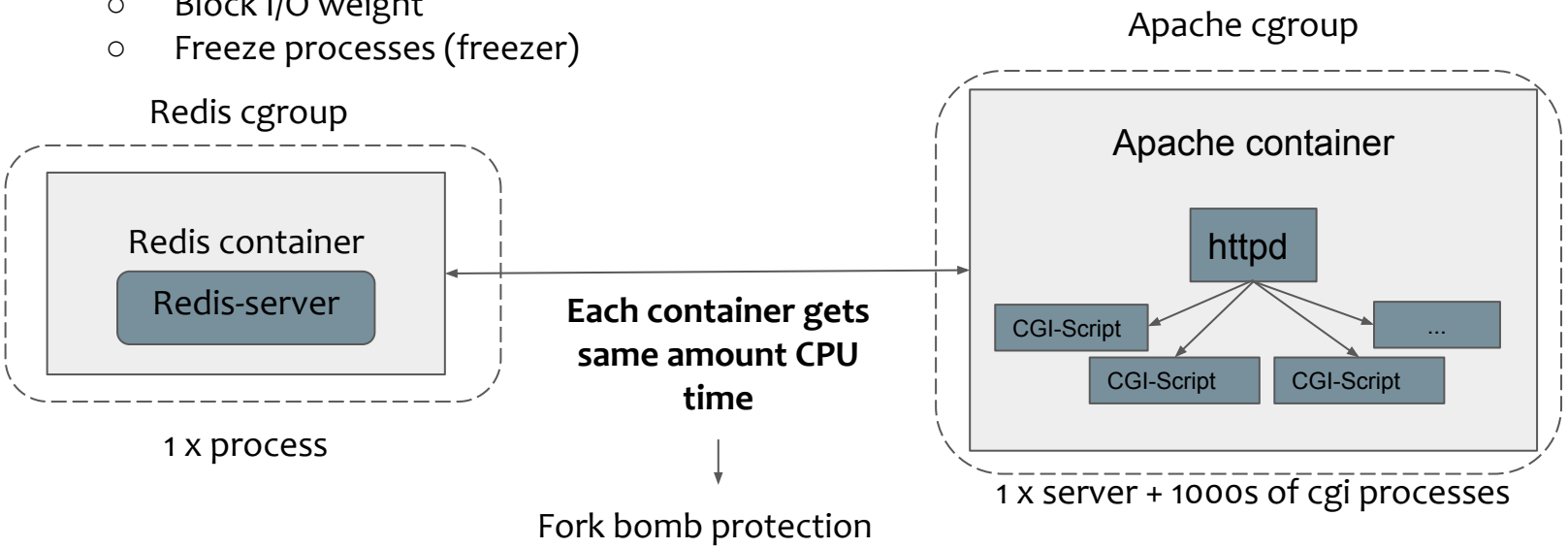
- Own routing table and set of network interfaces and firewall
- Often combined with veth-interfaces to provide virtual ethernet interfaces

User namespaces

- Allows to remap a user / group ids to different ranges
- Important for unprivileged containers (rootless), i.e.
  - Uid 1000 on the host becomes uid 0 (root uid) in the namespace

Cgroup

- Hierarchical resource groups:
  - Child inherits cgroup from parent
  - 9 different subsystems for different purposes
- Limits how much resources processes
  - Cpu schedule slices
  - Memory limits
  - Block I/O weight
  - Freeze processes (freezer)



Cgroup

- Also reliable process tracking:
  - SystemD/Docker puts each service/container in a cgroup
  - Member's of cgroup == all childs belonging to service/container
- Major API change: Cgroup v1 -> Cgroup V2
  - unified cgroup instead of separated by subsystem
- Interface through a filesystem: /sys/fs/cgroup/ (in this example cgroup v2)

Create cgroup
$ mkdir /sys/fs/cgroup/foo

Join a cgroup
$ echo $PID > /sys/fs/cgroup/foo/cgroup.procs

See current cgroup membership
$ cat /proc/self/cgroup

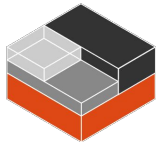
Seccomp/Capabilities/MAC

- **Goal**: restrict (root) user further by restricting system calls it can execute
- Capabilities
  - More granular permissions instead of privileged vs unprivileged processes
  - Examples:
    - CAP_CHOWN: change uid/gid of arbitrary files
    - CAP_NET_ADMIN: network configuration
- Seccomp/seccomp-ebpf
  - Filter programs that prevent system calls based on arguments
  - Example: disallow chown() with setuid bit set.
- MAC:
  - Application firewall: More complex policies/profiles to allow/disallow what files/resources
  - Two major implementations: apparmor or selinux

Implementations

Overview over implementations (1/2)

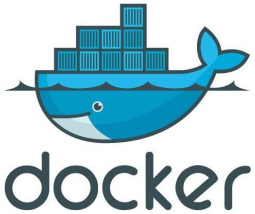
- “System” container engines (runs multiple services like a VM)
  - Systemd-nspawn, lxc/lxd, Openvz, …



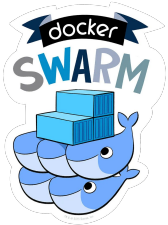
- Application sandboxes, special purpose container engines
  - Bubblewrap, snap, chromium sandbox
  - Singularity (High performance Computing)

Overview over implementations (2/2)

- "Application" container engines (one container per service)
  - **O**pen **C**ontainer **I**nitiative: Mostly standardized images/runtime
  - Docker, Podman, Containerd, …



- Cluster manager:
  - Schedule container over multiple hosts
  - make use of the underlying container engine



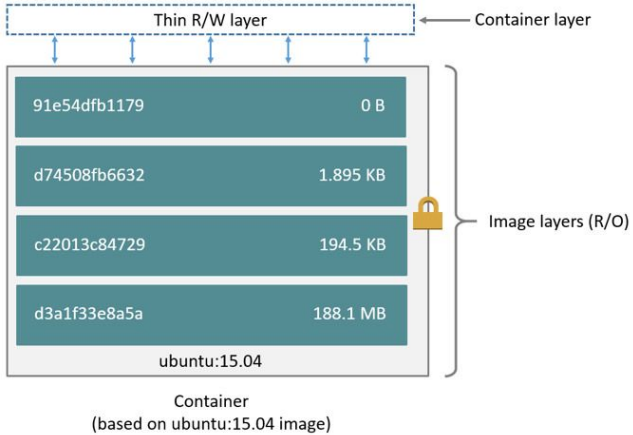
Used to run your tests!

OCI image & runtime

- **Docker:** Most popular container runtime
- **Dockerfile:** build description to build container images
- Makes heavy of overlay/snapshot filesystems to save space

 # Dockerfile*
FROM ubuntu:18.04
COPY . /app
RUN make /app
CMD python /app/app.py

$ docker build



- Standardized by OCI:
  - both runtime (i.e VMs instead of container) and build tools (buildah instead of Dockerfile) can be exchanged

* Dockerfile used in our course https://gist.github.com/Mic92/7ae91f5f0239acb56c67535c683f1bc1

Demo time!